University of Technology Darmstadt

Intellectics Group

TefoA – Testbed for Algorithms

Technical Report AIDA–03–10

Klaus Varrentrapp, Jürgen Henge-Ernst

klausvpp@intellektik.informatik.tu-darmstadt.de, juergen@henge-ernst.de

Contents

1 Introduction **1**
- 1.1 Document Structure . . . 2
- 1.2 Notes . . . 3
- 1.3 Thanks . . . 3

2 Testbed Design **4**
- 2.1 Experiments with Algorithms . . . 5
  - 2.1.1 Examples . . . 5
  - 2.1.2 Process Analysis . . . 7
- 2.2 Requirements for a Testbed . . . 11
- 2.3 Components of Experimentation . . . 13
  - 2.3.1 Integration and Specification of Algorithms . . . 14
  - 2.3.2 Problem Instances . . . 32
  - 2.3.3 Configuration and Experimentation . . . 32
  - 2.3.4 Statistical Analysis . . . 34
- 2.4 Data Management . . . 40
- 2.5 Architecture and Implementation . . . 42
  - 2.5.1 Implementation . . . 42
  - 2.5.2 System Requirements and Authentication . . . 43
  - 2.5.3 Database . . . 46
  - 2.5.4 Distribution of Jobs . . . 47
  - 2.5.5 Statistical Analysis . . . 48

3 User Interface Description **50**
- 3.1 Installation . . . 51
  - 3.1.1 System Requirements . . . 52
  - 3.1.2 Required Software Installation . . . 54
  - 3.1.3 Installing the Testbed . . . 61
  - 3.1.4 Configuring the Testbed . . . 69
- 3.2 Getting Started . . . 74
  - 3.2.1 Example Module . . . 74
  - 3.2.2 Installing a Module . . . 76
  - 3.2.3 Importing Problem Instances . . . 77
  - 3.2.4 Creating an Algorithm . . . 78
  - 3.2.5 Creating a Configuration . . . 80
  - 3.2.6 Creating an Experiment . . . 82
  - 3.2.7 Running an Experiment . . . 84
  - 3.2.8 Evaluating an Experiment . . . 85
- 3.3 Testbed in Detail . . . 95

3.3.1 User Input . . . 95
3.3.2 Submenus . . . 96
3.3.3 Problem Types . . . 103
3.3.4 Problem Instances . . . 104
3.3.5 Modules . . . 104
3.3.6 Algorithms . . . 107
3.3.7 Configurations . . . 110
3.3.8 Experiments . . . 116
3.3.9 Jobs . . . 121
3.3.10 Data Extraction . . . 125
3.3.11 Statistical Analysis . . . 129
3.3.12 Preferences . . . 132
3.3.13 Testbed Status . . . 133
3.3.14 Hardware Classes . . . 134
3.4 Command Line Interface (CLI) . . . 136
3.4.1 Extract Data . . . 136
3.4.2 Module Management . . . 138
3.4.3 Importing Data . . . 139
3.4.4 Starting a Job Server . . . 140
3.4.5 Maintaining the Database . . . 142
3.4.6 Display Job Results . . . 144
3.4.7 Display Data Structures . . . 145
3.5 Organizing and Searching Data . . . 146
3.5.1 Search Filters . . . 146
3.5.2 Categories . . . 165

4 Advanced Topics **171**
4.1 Quick Introduction to PHP . . . 172
4.2 Integrating Modules into the Testbed . . . 181
4.2.1 Module Definition File Generation Tools . . . 181
4.2.2 Basic Settings . . . 183
4.2.3 Parameter Definition . . . 187
4.2.4 Defining Performance Measures . . . 190
4.2.5 Adjusting the Execution Part . . . 190
4.3 Writing Data Extraction Scripts . . . 192
4.3.1 Table Format . . . 193
4.3.2 Commands and Predefined Variables . . . 195
4.3.3 Examples . . . 204
4.3.4 Further Information . . . 208
4.4 Writing Analysis Scripts . . . 212
4.4.1 Further information . . . 214
4.5 Web Interface for the Database . . . 215
4.6 Troubleshooting and Hints . . . 217

5 Architecture **227**
5.1 Database Structure . . . 227
5.1.1 Generation of Search Queries . . . 228
5.1.2 Design Issues . . . 231

5.2 Testbed Structure . . . 232
5.2.1 Applications and Services . . . 232
5.2.2 Templates . . . 234
5.2.3 Directory Structure of the Testbed . . . 235
5.2.4 Naming Conventions for Class Names . . . 238
5.2.5 Directory Structure of an Application . . . 239
5.2.6 Important Environment Variables . . . 241
5.2.7 Session Variables . . . 243
5.2.8 Global Functions . . . 245
5.2.9 Basic Classes . . . 250
5.2.10 Example . . . 265
5.2.11 Notes . . . 267
5.3 Extending the Testbed . . . 269
5.3.1 Using FORMs in UI . . . 269
5.3.2 Extending the Search Mask . . . 270

6 Future Work **271**

A Source Code **283**
A.1 Modules . . . 283
A.1.1 Example: Pruned Dummy . . . 283
A.1.2 A Wrapper for an Executable . . . 286
A.2 Data Structures . . . 289
A.2.1 Structure algorithms.soalgorithms . . . 289
A.2.2 Structure common.sohardware . . . 289
A.2.3 Structure common.soproblemtypes . . . 289
A.2.4 Structure configurations.soconfigurations . . . 290
A.2.5 Structure experiments.soexperiments . . . 290
A.2.6 Structure jobs.sojobs . . . 290
A.2.7 Structure probleminstances.soprobleminstances . . . 291
A.2.8 Structure statistics.soresultscripts . . . 291
A.2.9 Structure statistics.sorscripts . . . 291

B Glossary **292**

C Bibliography **294**

Index **301**

List of Figures

2.1 Work flow of experimentation . . . 7
2.2 Interfaces . . . 12
2.3 Components of experimentation . . . 13
2.4 Model of a module . . . 15
2.5 Model of an algorithm . . . 31
2.6 Module structure of PAM . . . 45

3.1 Main menu of testbed . . . 78
3.2 Selecting a problem type . . . 79
3.3 Creating an algorithm . . . 79
3.4 Creating a configuration: Entering basic information . . . 80
3.5 Creating a configuration: Setting the parameters . . . 81
3.6 Creating a configuration: List of fixed parameter settings . . . 82
3.7 Creating an experiment . . . 83
3.8 Starting an experiment . . . 86
3.9 List of jobs . . . 86
3.10 List of finished jobs . . . 87
3.11 Extracting data from job results . . . 88
3.12 Data extraction: Calculating columns . . . 89
3.13 Data extraction: Viewing results . . . 90
3.14 Analyzing data . . . 93
3.15 Analyzing data: View results . . . 93
3.16 Analyzing data: File listing . . . 94
3.17 Submenu appearance . . . 100
3.18 Creating a problem type . . . 103
3.19 Creating a problem instance . . . 105
3.20 Modules submenu . . . 105
3.21 Edit description of a module . . . 106
3.22 Detailed view of a module . . . 106
3.23 Algorithms submenu . . . 108
3.24 Creating an algorithm: Setting and hiding default parameters . . . 108
3.25 Configurations submenu . . . 111
3.26 Experiments submenu . . . 116
3.27 Detailed view of an experiment . . . 120
3.28 Jobs submenu . . . 121
3.29 Data extraction script submenu . . . 126
3.30 Creating a data extraction script . . . 126
3.31 Analysis scripts submenu . . . 130
3.32 Creating an analysis script . . . 130

3.33 Preferences . . . 132
3.34 Testbed status submenu . . . 133
3.35 Hardware classes . . . 135
3.36 Search filter: Generation mask imploded . . . 147
3.37 Dependencies between data types . . . 149
3.38 Search filter: Generation mask expanded – top . . . 151
3.39 Search filter: Generation mask expanded – bottom . . . 152
3.40 Search queries . . . 153
3.41 Categories submenu for experiments . . . 167
3.42 Detailed view of a category . . . 168
3.43 Add a category for experiments . . . 169
3.44 Setting current search filter from categories . . . 169
3.45 Assigning objects to categories . . . 170
3.46 Managing global categories . . . 170

5.1 Database Structure . . . 229
5.2 Object classes . . . 233
5.3 Directory structure of an application . . . 239

List of Tables

2.1 Parameters required for a module 16
2.2 Parameters strongly recommended for a module 16
2.3 Example command line interface definition output 22
2.4 Summary of the blocks of the standard output format 29
2.5 Example module output with proper standard output format 30

3.1 Directory names, naming conventions and abbreviations 50
3.2 Common icons and actions 99
3.3 Experiment statuses 117
3.4 Actions application to jobs 122
3.5 Job statuses 123
3.6 Actions for jobs: Icons and effects 124
3.7 Available object type for displaying their internal data structure 145
3.8 Wildcard examples 166

4.1 Example of a simple command line interface definition output 183

1 Introduction

Conducting computational experiments and analyzing their results in a sound manner can be tedious. Experiments have to be carried out, i.e. algorithms in various configurations with several inputs and repetitions have to be run. Results have to be analyzed from different perspectives, including a statistical evaluation. This document describes the usage and assembly of a testbed for conducting computational experiments with algorithms which automates recurring tasks.

The need for a testbed arose that allows to concentrate on the development of algorithms, in particular for Metaheuristics [42, 7, 8], instead of spending time with recurring (management) tasks such as:

- Storing and subsequently searching for data of various kind.
- Writing scripts to execute experiments.
- Statistical evaluation of the results.

Typical practice of computational experiments is to carry out a lot of experiments, i.e. algorithms are run, over time. The results of the runs are stored somewhere in the file system. Even done in an organized manner, a lot of time is required to find relevant data in the file system for one or more experiments, for example to do a statistical analysis based on some results. As nearly every scientist uses individual tools to conduct experiments, it is very difficult and almost impossible to share experimental results or to reproduce experimental results by other scientists. A testbed should help to reduce repeating work, aid in searching for data and enable sharing data for all relevant aspects of computational experiments.

Within the discourse of this manual, the most relevant aspects of computational experiments are identified that were used to guide the design and implementation of this testbed that meets the requirements thus identified. The focus is on algorithms as they are developed in typical computer science and computational intelligence fields such as machine learning, planning, and Metaheuristics. These algorithms have in common that they do not need interactive user interaction, work on input in the form of not too complex data structures and yield simple *performance measures* that typically are

of numerical nature and hence can easily be subject to statistical evaluation. Complex software systems with blurred performance measures or complex data structures as results are not addressed by the testbed. Further focus is on enabling intuitive and easy to use statistical evaluation and analysis of algorithmic results using existing statistical tools . Statistical evaluation and analysis comprises hypothesis testing, model building, and exploratory data analysis, in particular in graphical form through plots. The testbed is designed to elevate the practice of experimentation for computer scientists from the imperative level where researchers have to write a lot of scripts to carry out the various stages of experimentation to a declarative level where the researcher is only concerned with the specification instead of being concerned with the implementation of the experiments, too; the testbed implements the experiments as specified.

Note: The notion of 'experiments with algorithms' can on the one hand denote the analysis of the worst case runtime of an algorithm with the help of the O notation. On the other hand, it means the empirical analysis by means of actually running the algorithm under investigation on some problem instances. The latter meaning of experiments is also denoted by prepending 'empirical' or 'computational' throughout this text. The notion of an *experiment* in this document always denotes computational experiments instead of analytical experiments.

1.1 Document Structure

This document is organized as follows:

- Chapter 2: Testbed design
  This chapter contains a description and analysis of how experiments with algorithm are conducted and which requirements arise for a testbed supposed to automate this process. The design of the testbed including some interface specifications needed for incorporating arbitrary algorithms is presented next in this chapter.

- Chapter 3: User documentation
  A tutorial describes how to use the testbed to carry out experiments. The components of the testbed are described and it is shown in detail how algorithms can be used within the testbed, how experiments can be specified and run, how data can be searched for, and how data can be organized.

- Chapter 5: Advanced user documentation
  This chapter discuses the more intricate details of the testbed such as writing data extraction scripts and scripts for a statistical analysis. Additionally, it provides hints and contains a troubleshooting section.

- Chapter 5: Programmers documentation
  The implementation design and architecture of the testbed framework and the

database structure is explained in this chapter. This chapter also explains how new components and extensions can be added to the testbed and which are the important functionalities and components the testbed framework provides.

- Chapter 6: Future Work
  While developing a testbed like the one described in this discourse, constantly new ideas and possible approaches arise which can improve the usability and flexibility. Possible additional improvements are presented in this chapter.

1.2 Notes

This testbed can be obtained, used and extended under the GNU General Public License (GPL) (*http://www.gnu.org/licenses/licenses.html*) via the testbed's home page under [62]. The testbed is free to be extended by anyone who is interested in. Some possible extensions are listed in chapter 6 on page 271. Information how to participate is available via the testbed home page, too. The home page provides contact information for issuing comments, proposals for useful extensions, reports of bugs, and errors in the user manual. The testbed home page also is intended to be the place to exchange data extraction and analysis scripts.

1.3 Thanks

Thanks to Dr. Thomas St"utzle, Prof. Dr. Wolfgang Bibel, Ben Hermann, Stefan Pfetzing, Patrick Duchstein, Oliver Korb, Jens Gimmler, Mauro Birrattari, and Ulrich Scholz for their help and advice.

2 Testbed Design

This chapter describes the motivation to develop a testbed for experimentation with algorithms, next presents an analysis of the process of computational experimentation, and subsequently discusses the design of the testbed. This work is not intended to deal with the process of scientific experimentation in general as a common activity of many scientific disciplines such as physics, biology, chemistry, psychology, and so on. The tool developed is concerned only with the empirical experimentation with algorithms. Algorithms are delimited to be programs that can be taken more or less as black boxes, that yield a number of simple performance measures, and that do not need complex or even interactive user interaction. Whole software systems such as complex simulators or other complex programs requiring user interaction or yielding complex data structures as results are not topic of this work.

The process of computational experimentation [1] with algorithms is analyzed next and certain problems faced during this process are identified. In particular, recurring tasks and central aspects of experimentation with algorithms are pointed out. Starting from there, an analysis of the features a testbed intended to ease experimentation should incorporated is undertaken. This analysis will result in special notions identifying the main aspects of experimentation and thus conceptually structuring the process of experimentation. Next, based on the features needed by a testbed as resulted from the previous analysis, a design of a testbed providing these features is proposed and subsequently refined. The order of discussion in this chapter is as follows.

The first section of this chapter describes the practice of how experiments with algorithms are conducted. Next, the requirements for a testbed are identified and listed. Finally, the design and architecture of a new testbed for experimentation with algorithms is presented together with a treatment of the testbed implementation and some implementation specific important aspects of the testbed. The user interface is discussed in detail in chapter 3 on page 50.

[1] Experiments/experimentation with algorithms or simply experimentation in short

2.1 Experiments with Algorithms

In this section first some examples of how experiments typically are carried out are presented intended to point out how experiments with algorithms are performed. With the help of those examples the process of experimentation and the single features or rather components of experimentation with algorithms are identified and discussed.

2.1.1 Examples

In all examples described next, the experimenter has implemented an algorithm in the form of a program or rather binary executable [2], has defined a set of configurations and has created problem instances which should be used in the experiment.

An experiment for testing the influence of parameters of algorithms on their behavior, e.g. the influence of runtime on the solution quality, is typically conducted as follows. First, a script is written (e.g. Perl [65, 66], shell) to run the algorithm iterated over the parameter ranges of interest. While running the algorithm with various parameter settings a lot of output files are written into a directory. This step is repeated with different problem instances, each step possibly performed in another directory. Next, another script (e.g. Perl, awk) is written to extract and format the necessary data from the output files distributed over different directories. The data extracted is then analyzed with a statistical program like R, S-PLUS [60, 19, 20, 21, 22, 23, 24] , or plots are produced with plotting tools like Gnuplot. However, for each new analysis of a different aspect of parameter influence a new extraction script has to be written to collect the necessary data for the subsequent statistical analysis. Of course, a new script for the statistical analysis has to be created, too.

Another example of experiments with algorithm is the comparison of different algorithms for the same problem type. Each algorithm is run with a configuration for the algorithm's parameters on different problem instances. This has to be done manually or with a complicated script, because each algorithm has different parameter settings. Again, the results are stored in directories. Next, a script has to be written to extract the data needed from the different output files and transform it to a format needed for a subsequent statistical analysis.

Another frequent task in algorithm development is tuning of an algorithm, i.e. trying to find the optimal values for the parameters controlling the behavior of the algorithm in order to optimize the algorithm's performance. Running an algorithm with every possible parameter setting usually takes a prohibitively long time. Additionally, some parameter settings can be identified as being suboptimal in advance while other parameter settings might simply be not valid parameter settings for reasons of inter parameter

[2] Executable, binary, or program for short.

constraints. In order to speed-up the process of tuning, these parameter settings should not be used. Thus, algorithm tuning often requires to run the same algorithm with many parameter settings that can be quite complicated subsets of the set of all possible or feasible parameter settings. Scripts implementing the application of such complicated subsets to an algorithm can become quite big and complex, too; these scripts are likely to be hard to maintain, debug, or change. Extracting the precise values for certain parameter settings can be quite tedious.

Different fields of computer science and artificial intelligence have different practice of running algorithms. In case of metaheuristics algorithms typically only one algorithm (executable/program) is to be executed for a run on one problem instance. Planing algorithms typically run more than one program sequentially in a specific order. For example, first a problem instance generator generates the problem instance, then a preprocessing of the problem instance is performed, next a planing algorithm is run on the preprocessed data. Afterwards a post processing program analyzes the output from the main planing algorithm. For each step (preprocessing, planing and post processing) different algorithms (programs) are available. It is desirable to have these algorithms to be easily exchangeable. Hence, the complete algorithm for planing problems is not a monolithic algorithm, instead it is a sequence of modules where different modules can be exchanged.

Over time various experiments are executed and the results are stored someplace in the file system. The experimenter might want to get back to results and experiment settings of former experiments. For example, for a new experiment the user needs a special configuration of an algorithm which solves a special problem instance very fast. If the configuration was not remembered directly, the file system must be searched for until the configuration needed is found. Depending on the amount of experiments and how good the experiments have been organized in the file system, this can take a long time, especially if the configuration is hidden in a huge script file. If, unfortunately, the script which produced the configuration was deleted and perhaps only the output exists, the script must be written and tested anew to redo the run of the algorithm in the specific configuration.
When doing experiments with randomized algorithms, each algorithm has to run multiple times. Depending on the seed of the random number generator and hence depending on chance, the results of an algorithm will be different. A statistical analysis is mandatory to generalize scientifically sound from the results of the experiment to the general case. A prediction based on the results of just one run might very likely have no prediction power, so provision of multiple runs is vital.

2.1.2 Process Analysis

This subsection discusses and identifies the critical aspects of computational experimentation with algorithms. The following subsections then summarize the results and cast them into requirements for the design of a testbed intended to support computational experimentation with algorithms. In all cases discussed before some components of experimentation with algorithms are almost always present. These components are *modules*, *algorithms*, *configurations* (parameter settings for an algorithm), *problem instances*, *experiments*, *data extraction* and *statistical analysis*. This indicates that the process of experimentation includes some invariants.

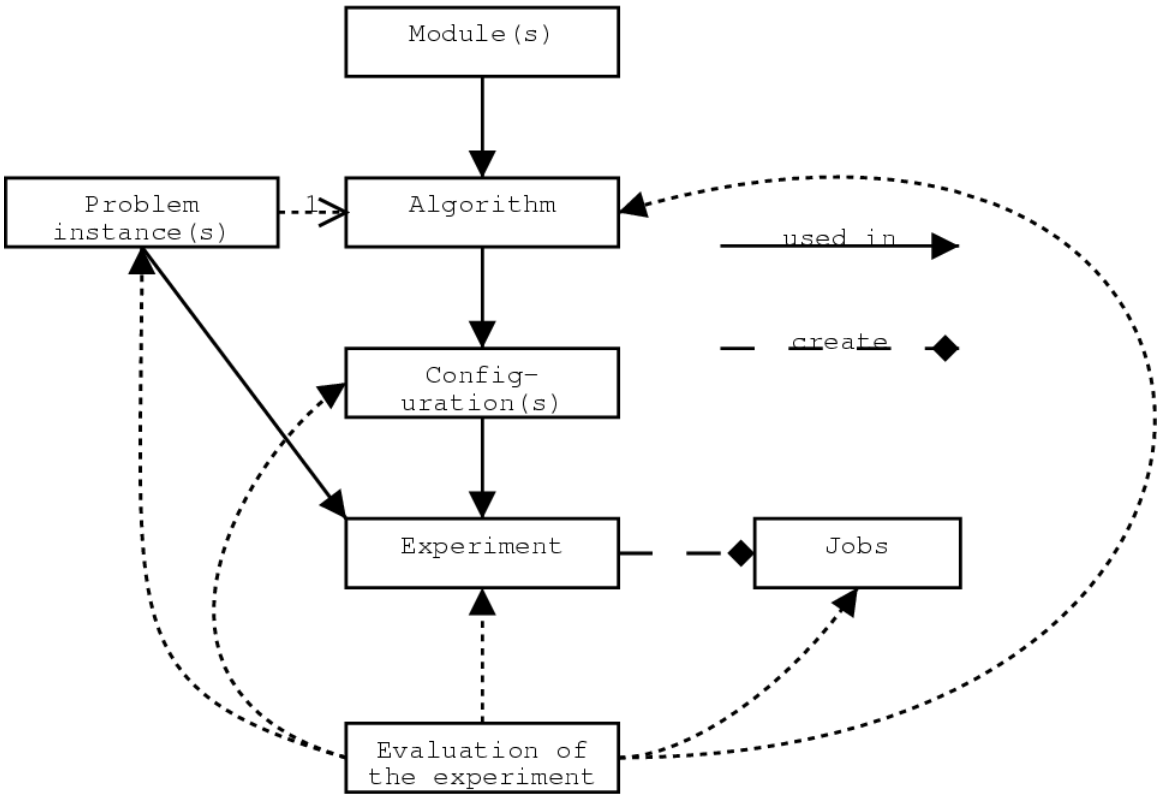


Figure 2.1: Work flow of experimentation

Figure 2.1 depicts the typical work flow of conducting experiments with algorithms.

First, some modules are combined to form an algorithm. Modules are executables that can be run via a command line command. The parameters needed by a module are provided as arguments to the command line call. As described in the example case for planning algorithms, algorithms in general can include per- and post-processing parts beside the main algorithm. These parts together are rather viewed as a single algorithm, even if this algorithms in practice is split into different modules (modules are discussed in detail in subsection 2.3.1 on page 14). So, although build from smaller components an algorithm as a whole still is the center of attention in experimentation; its set of

furnished parameters is the union of the sets of supported parameters of the modules the algorithm consists of, naming conflicts left aside for the moment.

Based on an algorithm a configuration of the algorithm is defined by setting values for the different parameters of the algorithm. In the course of experimentation, it is most often the case that not only one distinct parameter setting for an algorithm is tested but quite a lot of such settings. Each parameter for the algorithm can adopt multiple values. The algorithm can run in many configurations up to a *full factorial design* based on the sets of *parameter values*, i.e. all possible combinations of values for the individual parameters are formed similar to the Cartesian product of sets (see [17] for further information about experimental design).

Next, the algorithm in various configurations is run on a set of problem instances. The resulting set of runs to do, also called *jobs*, essentially make up an experiment. The jobs (one for each pair of problem instance and configuration) are then executed. The result of all jobs is finally analyzed by using the information used to create the experiment such as the various parameter settings and the information contained in the output produced by the job.

Two main experiment types have been identified. In the first type of experiments, a parameter combination for an algorithm is searched for which produces the best solutions in the shortest time. This case is called parameter tuning. For the second type different algorithms and configurations are compared to see how good the different algorithms and configurations solve a specific set of problem instances in comparison.

A lot of algorithms, for example metaheuristics, are randomized. Each run of an algorithm with the same parameters and on the same input file can produce different results. Hence, for obvious reasons, a statistical evaluation is needed to prove predictive results.

One goal of any tool is to help the user to automate recurring tasks. In the case of experimentation with algorithms such recurring tasks can be identified as being the specification of algorithms, configurations and experiments, the execution and supervision of jobs, and the final statistical evaluation of the results including retrieval of the necessary data and subsequent transformation of the necessary data in a format that can be processed by an analysis tool such as an statistics package. The user should not need to write scripts for the recurring task again and again.

Algorithms, i.e. sequences of individual modules in the form of programs, are written by different people. No standard has been established yet how parameters in the form of command line arguments for the program have to be named or used. Additionally, no standard is agreed upon how the output or result of an algorithm should look like. In order to integrate an algorithm into the testbed, an interface specification is needed which the algorithms must fulfill to be able to run inside the testbed.

The amount of work for the statistical evaluation can get huge as the data from the output must be extracted and transformed into a certain format (with a script or by

hand), next this data must be passed to an analysis tool, and afterwards the result of the statistical evaluation must be interpreted. Most of the process of statistical evaluation could be automated, if a standardized output format is used. In the case of adopting the format of existing statistical packages such as R, scripts implementing a statistical analysis in the package's programming language can be reused for the statistical evaluation of similar experiments. In case of R, such scripts can be called directly from within other programs. Such it is possible to completely and transparently integrate R into a testbed.

As mentioned before, algorithms do generally not follow a standard interface and each user has hence written individual scripts for execution of algorithms and for extraction of data from the results. It follows that it is nearly impossible to reproduce experiments without having all those scripts. The scripts typically follow the flavor of the experimenter and are not standardize, too. Working with scripts for purpose of execution of jobs of an experiment typically scatters the data of the experiments in the form of specifications of algorithms, configurations, experiments, jobs, results and so on in the file system. It is very difficult to extract information of a specific kind such as several algorithms specifications across different experiments, all problem instances used with an algorithm in a special configuration, or all jobs that run on an algorithm in a special configuration across multiple experiments. It is even harder to extract the associated results from jobs of such queries. Nevertheless, such queries frequently arise in the course of a statistical evaluation of algorithms.

Typically, an experiment is designed to answer a given question. The data is grouped and stored accordingly. If, based on old data, another aspect is of interest, this data sometimes can not be retrieved anymore. Instead of reusing old and perfectly valid results, new possibly tedious experiments have to be set up again.

Keeping all data centralized in one place, for example, a database, which links to the components used like the algorithms used in a configuration or configurations and problem instances used in an experiment, enables the user to manage *all* experimental data. A centralized data management in the form of a database can utilize all the functionality of a database which are evidently needed for management of experiments, too.

The usage of a database is not widely spread among empirical experiments with algorithms, because it is too much overhead for researchers to set up a database and conduct experiments accordingly. Having a testbed, however, the information and data need not be scattered over different scripts and output files any more. A centralized data management supports the standardized storage of data and hence makes exchange of experimental data feasible. Scientists can more easily reproduce mutual results, if import and export facilities are provided, too.

Any efficiently usable software system needs an efficient user interface. Graphical user interfaces (GUI) have shown to be more appealing than command line based user interfaces to most user. If designed properly, the efficiency loss compared to command

line based user interfaces can be diminished or even reversed. The user interface should be aligned to the work flow of experimentation to intuitively guide and support the user through and with the process of experimentation. In case of the testbed, it seems desirable to use a graphical user interface and to device its structure according to the central components of the experimentation process to achieve the characteristic feature. Finally having a web based user interface enables the user to access a testbed remotely without installing separate software on the client computer. A web based GUI can be used locally, too, providing a complete GUI on any local machine, too. Such a GUI can also be used in a multi-machine setup.

As the examples of subsection 2.1.1 on page 5 indicate, the process of experimentation still is handled in an imperative manner by writing scripts. As in the field of programming languages it is desirable to have a shift from imperative specification of experiments with algorithms to a declarative form which can unify the specification of the various stages of this experimentation process. A testbed for algorithms should enable exactly this by taking off the burden of dealing with the practical details of running the algorithms, managing the data, and implementing and running the analysis. The user just specifies what has to be done instead of programming it; the testbed is taking care of the details such as proper storage and retrieval of data, execution and supervision of executables and extracting and transforming results for the statistical analysis.

2.2 Requirements for a Testbed

Based on the analysis of the preceding subsection 2.1.2 on page 7 and general requirements for software systems, the most important requirements for a testbed for computational experimentation are identified as:

1. Automation of recurring actions while running experiments. This implies including specification facilities for all aspects of experiments and the supervision of execution of jobs including recovery on failure. Experiments should be specified in a declarative manner instead of an imperatively way.

2. Existing modules and algorithms should be able to run in the testbed. This can be accomplished by providing a standard interface for running modules on the command line level. Existing modules can be made compliant to this interface by *wrapper* construction

3. Enabling the construction of algorithms by sequencing modules.

4. Provision of centralized storage of *any* data related to the process of experimentation and provision of sufficient search and management facilities for data and results of any type within the testbed because any data or information is possibly interesting for an experimenter. This should be enabled with an easy to use and intuitive search tool. Provisions to support multi-user and multi-machine modes should be considered. Altogether, this strongly indicates the employment of a database management system to stored *all* the data of the testbed.

5. Provision of means for statistical evaluation of all experiments that were run within the testbed. This necessitates the specification of an output format for jobs and a data extraction language for flexible extraction of data from the results. This also includes proper integration of existing statistic packages, possibly by providing facilities to run scripts describing statistical evaluation for these packages within the testbed.

6. Usage of a graphical user interface to manage all aspects of the testbed. The user interface design should follow the work flow of experimentation to guide the user through the process of experimentation. It preferably is web based and multi-user and multi-machine capable.

7. Enabling exchange of any information and data in one testbed with another testbed installation preferably in an human readable format. This includes import and export facilities for *any* aspect and data type.

8. Platform independence. The testbed should primarily run on Linux but should also run on any other Unix (POSIX standard) systems.

9. Easy extensibility to meet new needs.
10. No fees for licenses to use the testbed, since it is to be used in an academic environment.

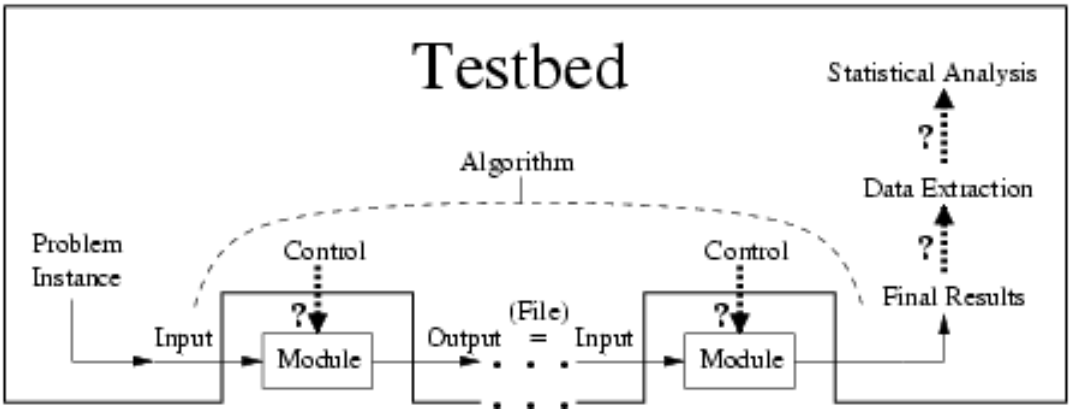


Figure 2.2: Interfaces

In order to fulfill the pivotal points of these requirements, several interfaces have to be devised as is depicted in figure 2.2 with question marks:

- Running the algorithms: How can the algorithm be controlled? What has the command line interface of an algorithm to look like? Can or even should wrapper be employed to provide more flexibility in integrating algorithms?
- Output format: How can the output of an algorithm be further processed in an automatic way? Is a standardized output format of some kind required to ensure at least some automation, in particular with respect to the subsequent statistical evaluation?
- Analysis: How can recurring tasks in analyzing algorithm performance such as creating plots, conducting statistical tests be performed?
  Two issues do arise here:
  1. Data extraction from algorithm results: How can the information of interest be extracted automatically, given that some standard with respect to the expected "raw" output is given (previous point)?
  2. Statistical analysis: How can the information of interest extracted such be further processed statistical methods. In practice, this winds up to the question which existing statistical tools to use and how to adapt to their input requirements.

Later parts of this manual will cover these issues in detail and will present the solution to these problems as taken by the testbed. The next section addresses the first two interfaces and how the testbed deals with them.

2.3 Components of Experimentation

The most important aspect with respect to the design of a testbed for conducting, managing and supervising computational experiments is how to reflect the process of experimentation in a natural way and how to model the various concepts emerging in this process. Accordingly, the designed structure of the testbed is centered around the work flow and notions of experimentation as depicted in figure 2.3 where the general procedure of conducting experiments is outlined. A crucial part of the testbed design is concerned with the integration of tools for statistical analysis of experimental results. The integration of a statistical package, in the case of this testbed R [60], has been accomplished by the definition of an output format for the result of jobs and the development of two scripting languages. The *data extraction language* enables generic extraction of specific information from job results and enables transformation into a format that is readable by the statistics package R. The testbed's R scripts facility provides generic access to all elements of the R language directly from within the testbed. Another crucial part of the testbed design is the integration of existing binary executables that implement algorithms. These binaries must heed some minimal interface restrictions in order to be able to be executed by the testbed. Additionally, the testbed provides tools for almost automatic integration of binaries, provided they comply to certain interface restrictions.

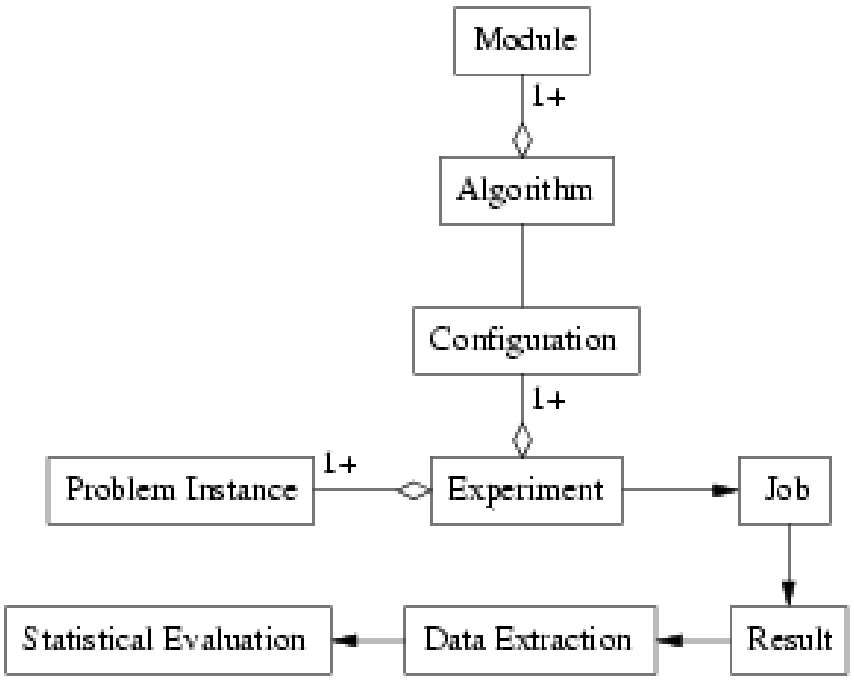


Figure 2.3: Components of experimentation

The main components of the experimental work flow are described here together with a simultaneous discussion of how these components can be integrated into the testbed and how they are enabled to work together. This section covers in detail the components of

experimentation, the various interfaces that enable a smooth cooperation of the components (marked with an ? in figure 2.3 on the preceding page, the scripting languages and the centralized data management employed.

2.3.1 Integration and Specification of Algorithms

An algorithm consists of a fixed sequence of one or more modules as illustrated in figure 2.5 on page 31 and figure 2.2 on page 12. These modules can be pre- and post-processing modules, the main algorithm module and so on. Each module gets its predecessors output as input. The first module's input is the input file for the whole algorithm and the last module in the sequence writes the output file for the algorithm as a whole. The data stream between the modules is realized via temporary files.

In what follows the integration and specification of modules and algorithms into the testbed is addressed. The discussion comprises the various interfaces that should, sometimes must be complied to. The two main interface formats the testbed requires are the output format the modules should adhere to when run last in an algorithm's module sequence to enable subsequent statistical analysis of the results and the command line interface used to address the module binaries on the command line which is mandatory. An optional extension to this interface is the provision to output the specification of the parameters and arguments for doing so.

Modules

A module is defined as being part of an algorithm. Only modules actually exist as binary executables and hence can be executed. In the following, the notion module in used to refer to the meaning of being part of an algorithm as well as to the meaning of being the executable binary that implements this part and which can be executed by the system by calling it with parameters. Sometimes, in order to make things clearer, the latter meaning is denoted by (module) executable, too. A module is supposed to expect its input via an input stream realized as a single input file and outputs its computation results into a single stream, again realized as an output file. The information about which files to use in addition to other information needed by the module is passed as parameters. Parameters are represented as command line arguments on the level of the executable. Figure 2.4 on the next page illustrates this procedure. A module can be viewed as a black box, which the testbed executes with given parameter values and provision of a proper input file and which returns a file for further handling by the testbed.

As pointed out before, in order to run arbitrary modules by the testbed and in order for the testbed to establish the flow of data from a single input source sequentially through a sequence of modules that form an algorithm, each module must heed a common interface.

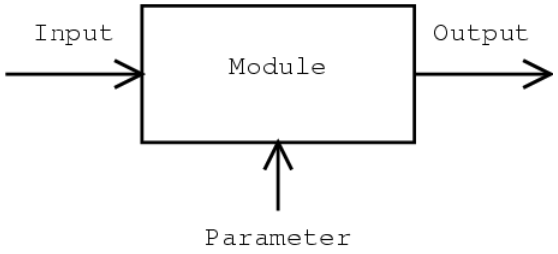


Figure 2.4: Model of a module

This common interface for modules consists of:

- Restrictions with respect to a required minimal set of parameters to be supported by each module,
- restrictions with respect to types and names of parameters,
- the optional but recommended requirement that each module must output its individual parameters information, called the *command line interface definition* of the module, on demand, and
- optional but recommended restrictions with respect to the format of the output each module eligible to be run last in a sequence of modules forming an algorithm should heed.

The first three restriction comprise the *command line interface*, the last restriction comprises the output interface of a module. These interfaces and their requirements are described in detail next. A pictorial view is presented in figure 2.2 on page 12.

Parameter Specification of Modules

Each module must be able to be called with a Unix like command line syntax called *command line interface definition format* during this discourse. A command in Unix consists of a list of strings. The first string represents the name of the executable. The next strings form a list of parameters. In case of the testbed the executables are the module executables and each parameter actually is represented as two arguments in the form of two string: One flag indicating the identity of the parameter followed by a value for this parameter. The flag can be a minus sign directly followed by a single letter, called *short flag*, or it can be two minus signs followed by any string, called *long flag*. Only 30 characters are assumed to be significant for the long flag and the testbed will only store at most 30 characters. Any parameters are set by the testbed using the long flag. The long flag excluding the leading two minus signs are assumed to be a parameters name. Any module must not have two identical names, modulo the 30 character limit,

of course. The short flag is only used by the testbed if no long flag an be found for a parameter. The value of a parameter can be an arbitrary string and need only make sense to the module itself.

The requirements for parameters basically consist of the syntax requirements explained just now and a minimum set of parameters any module must support. This minimal set of parameters is listed in table 2.1.

No.	Parameter	Range	Long Short Flag	Long Flag
1	Input	File Name	-i	--input
2	Output	File Name	-o	--output

Table 2.1: Parameters required for a module

No.	Parameter	Range	Short Flag	Long Flag
3	Help Request	–	-h	--help
4	Maximum CPU Time	Time in Seconds, Positive Integer	-t	--maxTime
	or	*or*		
	Maximum Number of Iterations	String representing Arithmetic Expression	-p	--steps
5	Number of Repetitions	Positive Integer	-x	--maxTries

Table 2.2: Parameters strongly recommended for a module

In order to run properly, any module must be called with at least these two mandatory parameters. All other parameters can be omitted. Generally, if a parameter is omitted when calling the module, the module uses its implementation dependent default values which must be provided by a module for any parameter it supports, except the two mandatory parameters for input and output.

In addition to these two mandatory parameters, some parameters are strongly recommended. Algorithms not necessarily stop after a fixed amount of runtime. Some can, in principle, run forever. Therefore, modules implementing such algorithms need to provide parameters for limiting the runtime in the form of a limited amount of time the module is supposed to run or a maximum number of some kind of steps it is allowed

to perform. As a lot of algorithms are randomized in addition, they need to be carried out for multiple repeated runs on a problem instance. Hence, parameters setting the number of repetitions or tries are needed, too. Table 2.2 on the facing page proposes a standard for these frequently needed parameters.

The testbed must be able to retrieve information about which parameters a module executable expects together with some name, typing, default, and description information in order to properly run the executable on the one hand and in order to display this information to the user when creating an algorithm or when configuring one on the other hand. All information related to the parameters of a module are conveyed to the testbed via a wrapper called *module definition file*. This file in special PHP syntax can be generated automatically by the testbed for a module, given the module complies to the optional requirements described in the following (see subsection 3.1.3 on page 61 and section 4.2 on page 181 for more information about module definition files). Otherwise, the module definition file must be constructed manually.

One optional requirement of the testbed's common interface for modules of the testbed is the provision of an output of the module specific *command line interface definition* or specification. Each compliant module must output all necessary information about its command line signature, i.e. its kinds and numbers of supported command line parameters, on standard output upon calling it with the help request: `--help` parameter flag coming as first parameter. The command line interface definition defines the parameters supported by a module by defining the flags, the ranges for parameter values, the module internal default values the module use when a parameter is omitted in the call, and a short description for each parameter supported. The parameters are defined in a block bracketed by key words `begin parameters` and `end parameters`. Each non-empty line in this block represents one parameter. Table 2.3 on page 22 shows an example of a command line interface definition output with each parameter occupying one line. A call for the example dummy module could be:

```
./Dummy --input "In.dat" --output "Out.dat" --maxTime 1200 --tries 30
                          --yMin 10 --yMax 1000 --seed 12345 --function 3
```

In addition to specifying the module specific parameters information about the output a module produces if it is the last module of an algorithm can be included into the command line interface definition, too. Basically, the module specifies which performance measures it will output.

More specifically, the command line interface definition format of modules is defined as follows:

- Comments: When reading a `#`, the rest of the line can be ignored
- Brackets `begin comment`, `end comment` frame plain text that serves as a comment for the module. Together with the module name and some additional description

the user can enter when automatically creating a module definition file (compare to subsection 4.2.1 on page 181), this information is displayed to the user when the user is choosing modules to define algorithms (see specification of algorithms on page 31)

- Brackets `begin performance measures`, `end performance measures` surround the list of performance measures a module computes, i.e. the list of final output information. Each module that can be run as the last of an algorithm declares its performance measure here. Each performance measure is listed in its own line and must have the format

  `<name> <type>`

  where `<name>` and `<type>` are separated by at least one whitespace, `<name>` is the name of the performance measure without any whitespace and `<type>` is a type from the set of types usable for defining parameter types (including `NO`), but without any subrange restriction. Anything after `<type>` is ignored. The type information for performance measures will not be checked or used by the testbed itself. A sample performance measure declaration part of a command line interface definition can look like:

```
begin performance measures
   best        REAL
   length      INT
   steps       INT
   counterForX INT
end performance measures
```

  The performance measures listed here will be automatically provided when calling a data extraction script on any result produced by an algorithm where this module is last in the module sequence and hence produces the algorithm output. See section 4.3 on page 192 for more details about writing data extraction scripts.

- Brackets `begin parameters`, `end parameters` encapsulate the list of supported parameters. Each parameter is specified in *one* line as a white space separated list of the following items: `ShortFlag`, `LongFlag`, `Range`, `DefaultValue`, and `Description`.

  - `ShortFlag` must be `-<character>` where `<character>` represents any single character. The short flag is of informative nature only, since it is not used by the testbed. It should not be omitted, though, since otherwise when generating the module definition file automatically, the following items will not be recognized correctly, e.g. the long flag becomes the short flag and so on. Two or more identical short flags are allowed.

– `LongFlag` must be of format `--<string>` where `<string>` is any string of characters. As the user normally only can guess the meaning of a short flag, the long flag can be used to describe the nature of the parameter more descriptively. Additionally, without the two leading minus signs, the long flag serves as parameter name. The long flag is used by the testbed to set an parameter on the command line. With respect to automatic generation of a module definition file,the following information is important: Only characters are 'a' - 'z', 'A' - 'Z', '0' - '9', and '-' are allowed. Any not allowed characters are deleted. Only the first 30 characters after the two leading minus signs are significant.

  Examples:

  * Long flag definition `test` yields
    name 'test' and
    parameter call `--test`.
  * Long flag definition `-test` yields
    name 'test' and
    parameter call `--test`.
  * Long flag definition `--test` yields
    name 'test' and
    parameter call `--test`.
  * Long flag definition `---test` yields
    name '-test' and
    parameter call `---test`.
  * Long flag definition `--123456789012345678901234567890____test` yields
    name '123456789012345678901234567890test'
    and parameter call `--123456789012345678901234567890test`.
  * Long flag definition `--123456789012345678901234567890test` yields
    name '123456789012345678901234567890' and
    parameter call `--123456789012345678901234567890`.

– `Range` is defined as `TYPE` or `TYPE:SUBRANGE`:

  * `TYPE` can be either `REAL, INT, STRING, BOOL, FILENAME`, or `NO`, the first four having the same meaning as the identically named types in common programming languages. Type `FILENAME` essentially is a string, too, but is used to convey the additional information that the parameter is addressing a file. All parameters with type `NO` will be completely ignored by the testbed. This type is used for defining a `--help` parameter. When automatically generating a module definition, the type and any subrange restriction is translated into a regular expression which is used for checking the user input when actually defining values for a parameter as described in subsections 3.3.6 and 3.3.7 about setting parameters for

algorithms and configurations on pages 107 and 110. Additionally, the default value specified in the parameter definition is checked, too.
Types `REAL` and `INT` have to be encoded in conventional floating point representation, not in scientific notation. That is, `123.456`, `-123.456000`, and `000123.456` are valid values for a parameter of type `REAL`, while `1.23456e2` is not.
Parameters of type `STRING` or `FILENAME` expect a string as input. Note that if setting a parameter value in the testbed when configuring an algorithm, double or single quotes will be regarded as ordinary characters without any special meaning. Only the comma to separate strings is a special character. It can be escaped by a preceding comma (compare with subsections 3.3.6 and 3.3.7 on pages 107 and 110). Note that filenames must contain path information when given through configurations of the testbed (see subsection 3.3.7 on page 110 and the according entry in the troubleshooting list in section 4.6), otherwise the files will not be found since it they are looked for in a temporary directory created by the testbed on demand.
Parameters of type `BOOL` expect as input only value of two categories for either true of false. Note that such parameters expect a flag *and* a value, too. Parameters of type `BOOL` are not switches that are turned on or off according to whether the flag is set or not!

* The optional subrange `:SUBRANGE` restricts the range of a type `TYPE` to certain values.

  · Subranges for types `REAL` and `INT` can be (having the intuitive meaning):

    [a,b], (a,b),
    [a,b), (a,b],
    $>= a$, $< a$, $<= a$, and $> a$.

    Variables a and b are representing a real value in conventional floating point notation or an integer value.
    Only subranges $<= 0$, < 0, > 0, and $>= 0$ can be translated into appropriate regular expressions. All other subranges for numerical types are translated into a regular expression that checks, whether an input is a proper real value in conventional floating point notation or an integer value, respectively.

  · A subrange for types `REAL`, `INT`, `STRING`, and `FILENAME` can be an enumeration:

    i_1,i_2, ... ,i_n, with i_j being of type `TYPE` ($j \in \{1, \ldots, n\}, n \in \mathbb{N}$).
    Again, only the comma is a special character which can be escaped by a preceding second comma. The leading and trailing curly brack-

ets must not be omitted. These are deleted before parsing the enumeration, any other curly brackets are treated as normal characters without any special meaning.

· A subrange for types `STRING` and `FILENAME` finally can be a regular expression:

`/regular expression/modifier`

The regular expression follows the Perl and PHP syntax for regular expressions (see PHP-manual 'Regular Expression (Perl Compatible)', [54] for information about the syntax and semantics of the regular expressions expected here). Expression modifiers are `modifier` allowed, too.

∗ Examples of valid ranges are:

```
INT:
INT:[0,10]
INT:(-100000,100000)
INT:>100000
REAL:[0,1]
REAL:{0.1,0.2,0.3}
REAL:<-00.00
STRING:{"one",two,three,,four}
STRING:/.*test.*/i
FILENAME:{1.out,.stat,'.rest'}
BOOL
```

The first enumeration subrange for type `STRING` yields enumeration elements `"one"`, `two`, and `three,four`, the second enumeration subrange for type `FILENAME` yields enumeration elements `{1.out`, `.stat`, and `'.rest'`. The regular expression for type `STRING` only accepts inputs that end with `test` modulo upper/lower case.

– `DefaultValue` indicates the parameter value used, if the parameter has not been not set upon module call. The value must not contain any whitespace, of course. When automatically generating the module definition file, the default value will be checked if it complies to the type and possible subrange defined for the parameter, too. A default value `none` indicates that no default value is given or applicable. This can be the case, if the module does not depend on a parameter value and can do fine without, perhaps falling back to some internal default computation which can not easily be triggered by a special parameter value.

– `Description` gives a brief explanation of the parameter. This description is presented to the user whenever parameters for this module have to be set or chosen.

```
Call:
   Dummy  [-h|--help] [-i|--input]
          [-o|--output] [-v|--tries]
          [-s|--seed] [-t|--maxTime]
          [-m|--minTime] [-y|--yMin]
          [-a|--yMax] [-n|--maxMeasures]
          [-f|--function] [-i|--randomTime]
          [-r|--randomY] [-e|--randomType]
          [-l|--finallyFail] [-p|--reallyWait]

begin parameters
-h --help        NO                   Get help
-i --input       FILENAME    'a.tsp'  Input-file
-o --output      FILENAME    'IN.out' Output-file, 'IN' = name of input file
-x --tries       INT:>0      1        Number of tries (=repetitions)
-s --seed        INT:        0        Seed for random number generator
-m --minTime     REAL:>0     0.01     Time points of measurement in (sec)
-t --maxTime     STRING      'n'      Maximum netto CPU time per try (sec)
-y --yMin        REAL:>=0    0        Start of measurements
-a --yMax        REAL:>0     10000    Stop of measurements
-n --maxMeasures INT:>0      50       Maximum number of measurements
-f --function    INT:{1,2,3} 1        Function used
                                          # 1 => f(x)=1/x + yMin
                                          # 2 => f(x)=1/x^2 + yMin
                                          # 3 => f(x)=1/ln(x) + yMin
-i --randomTime  REAL:>=0    1        Degree of randomization of time points
-r --randomY     REAL:>=0    1        Degree of randomization of measurements.
-e --randomType  BOOL        TRUE     Probability distribution
                                          # Flag set     => Uniform
                                          # Flag omitted => Gaussian
-l --finallyFail BOOL        FALSE    Exit with exit code unequal to zero?
                                          # Failing does not affect output
-p --finallyWait BOOL        FALSE    Wait for additional maxTime seconds?
end parameters

begin performance measures
   best         REAL
   worst        REAL
   steps        INT
   stepsBest    INT
   stepsWorst   INT
end performance measures

begin comment
 This algorithm is intended for testing purposes for the testbed for algorithms.
 It simulates the behavior of a Metaheuristic used to solve combinatorial
 optimization problems by producing output similar to what would be
 produced by the Metaheuristic. The dummy also gives an example for a
 proper application of the command line interface definition format and
 the standard output format.
end comment
```

Table 2.3: Example command line interface definition output

Any module that does not meet the mandatory command line interface definition requirements as described in this paragraph must have an additional wrapper beside the module definition file to comply. Reasons why a wrapper may be needed are:

- Multiple input files,
- missing flags,
- different incompatible output format,
- parameters that only consist of a flag information,
- different incompatible names of some flags, or
- missing capability to perform repeated measures.

The testbed's internal wrapper for a module, the module definition file can be customized manually and hence can serve as a starting point for additional wrapper construction. A separate wrapper, however, can be constructed, too. For new modules that comply with all interface restrictions a module definition file, can easily be generated automatically as mentioned before. Typically, the generated file need not be modified anymore. Most problems and error with respect to parameter definition are detected and reported by the generation tool. See section 4.2 on page 181 for more information about module definition files, wrapper and wrapper construction.

The testbed installation contains a dummy module written in C (see subsection 3.2.1 on page 74) that implements some basic routines for parsing the command line call according to the command line interface definition format. These functions can be reused by means of copy & paste. Additionally, in the compressed `tar` file `DOC_DIR`*/examples/modules/Interfaces-Tools.tgz*, three classes written in C++ named `Parameter` and `ProgramParameters` (files *Parameter.h*, *Parameter.cc*, *ProgramParameters.h*, and *ProgramParameters.cc*) and are available that implement a convenient specification and parsing method for the command line interface of programs according to the command line definition format. These can be reused, too. For more information about these classes and the dummy module see the documentation and the comments in the code. Class `StandardOutputFormat` in the compressed `tar` file (files *StandardOutputFormat.c* and *StandardOutputFormat.h* implements a convenient method to output results in proper format according to the standard output format of the testbed (see paragraph 2.3.1 on the next page of subsection 2.3.1 on page 14). File *Interfaces-Tools-Example.cc* implements a demonstration of how to use the interface tools. All other files in the compressed `tar` file are auxiliary classes or files. It can be compiled using command `make`. All other files in the compressed `tar` file are auxiliary classes or files such as *PerformanceMeasure.h* and *PerformanceMeasure.cc* implementing a class to represent performance measures, *RandomNumberGenerator.h* and *RandomNumberGenerator.cc* implementing a random number generator, *Timer.h*

and *Timer.cc* implementing timing functionality, and *FatalException* and *Warning* implementing ways to issue warnings and fatal error messages that automatically end the program in case as error has occurred. All files are documented using the format of the Doxygen documentation system [37].

Standard Output Format

The output of a job, i.e. the output produced by an algorithm run in a fixed parameter setting on a specific problem instance, is finally produced by the last of the module the algorithm consists of. This output encodes the results of the run. The results typically comprise some final values of one or more performance measures, solutions and information about the behavior of the algorithm during runtime, for example the development of one or more performance measures or candidate solutions during runtime. The format for the output is standardized for the testbed and called *standard output format*. It extents the format of the Metaheuristic network [9]. Standardization of job output is necessary to enable the introduction of a data extraction language within the testbed. Such a data [3] extraction language enables easy extraction of arbitrary information from the results of jobs and thus enables translation of the output data into a format a statistics package or plotting program can process.

Before plunging into the details of the standard output format, some notes have to given. The format that is presented in what follows finally is only a proposal for how an algorithm's output could look like in order to enhance subsequent processing, in particular by the testbed. This proposal tries to bring some order into to the vast variety of conceivable output formats and tries to harmonize them a little bit to a common denominator with the goal to standardize and automate the data extraction process. In principle, however, the data extraction of the testbed via data extraction scripts is not confined to the standard output format that is presented here. Any other format can be processed and the relevant data can be extracted, too, since the data extraction scripts essentially are PHP programs. See the PHP manual [54] for an introduction to PHP.

[3] The notions of data and information as used here are as follows: Information is an abstract for the contents and subject of communication. Any information can have different meanings to different person. In order to communicate information it has to be encoded by some coding scheme on some carrier. Data is encoded information. Data can not exist without a carrier. Carriers can be sound waves, hard discs, RAM, pictures, and so on. During communication the sender encodes the information to be communicated with some encoding scheme on a carrier as data, for examples as a sequence of bytes that form a file on a hard disc, which then is transported to the receiver and decoded to form information again. The meaning of the encoded information may be different for sender and receiver, for example, if decoded differently than it was encoded in case of a misunderstanding. Results are information, too. In short: Data is encoded information or rather data represents information. Information is an abstract while data is in some sense a physical entity. Consequently, data extraction here really denotes extraction of parts of files. The type of data here relates to the type of the information it encodes. However, the distinction between information and data often is blurred. Thus, the notions information and data are sometimes used interchangeably here.

Unfortunately, some pre defined language constructs which are designed to take over and automate lot of tedious work when writing data extraction scripts are not usable if the standard output format is not followed. In such a case, each script has to be written anew. This requires some knowledge about programming in PHP. In the end, the actual output format being processed by an extraction script does not matter as long as the data extracted has been cast in a format similar to tables in relational databases which is the format used to interface to statistics packages and plotting programs. This format is described later when dealing with the internals of writing data extractions scripts (see subsection 4.3.4 on page 208).

In order to design an output format and accordingly in order to design the data extraction language for this format, it necessary to consider what types of results are of interest for an analysis and thus have to be extracted. The types of results typically of interest when analyzing algorithms can be depicted as a hierarchy. This hierarchy is utilized to design the output format such that conceptually different types of results are separated into different parts of the output.

Recall that the testbed requires provision for independent, repeated *runs* or *tries* (see section 2.1 on page 5 and section 4.2 on page 181). The data in the output of a job can be divided into *try independent* and *try dependent* data, i.e. data that is globally valid for the whole output and data that was produced by individual tries. For example, information that does not change from try to try is the command line call, the parameters used together with their assigned values, information about the problem instance used such as instance size or global optimum in case of an combinatorial optimization problem, and so on. Parameter information often is of particular importance. Note that the command line call need not include all parameters available as omitting parameters results in using module internal default values which nevertheless might be of interest. In this discourse, parameters can comprise any information about the idiosyncrasies of the run of a job. Parameter data can also contain information which is derived from other parameter information and hence can be redundant in principle.

Try dependent data, for example, holds information about the values of performance measures, the seed used for initializing the random generator for a try, and the solution computed during the try. Try dependent data can further be subdivided into two subcategories. The first category comprises data that describes the development or behavior of the algorithm during runtime. This behavior or development can, for example, be depicted by recording the development of performance measures and (partial) candidate solutions during runtime: whenever a new best solution or a new value of a performance measure has been found, it is output together with some time information. The second category of try dependent data typically is encompasses the final results in the form of final values of performance measures and encodings of final solutions.

The testbed's output format and the data extraction language is designed to act upon the hierarchy of output data. If the testbed's standard output format is heeded by the

last module of an algorithm, the testbed can automatically extract the data indicated previously. Extraction of any other type of data is possible, too. However, the degree of automation decreases.

The different parts of the output for the different types of data are separated by so called *blocks*. Blocks are indicated by *brackets* . The *opening brackets* of a block always begin with `begin` while the *closing brackets* always begin with `end`. The generic format of brackets is:

`begin <name> <value>`

contents

`end <name> <value>`

Note that there are no additional white spaces[4] allowed at the end of the line of the `begin` and `end` brackets. Additionally `<name>` *and* `<value>` must not contain any whitespace, too, since they are separated from each other exactly by whitespaces! Each unique bracket (unique `<name>-<value>` pair) may only be used once in the output. Certain kinds of data such as information about parameter settings, results, and other general information are supported directly by the testbed. This data is separated into blocks enclosed by brackets with reserved names and can be extracted automatically. Any information valid only for a special try has to be enclosed in brackets that follows the generic format described just now with `<value>` being the identifier or rather number of the try the data stems from. It must be a nonnegative integer. The name of the brackets, i.e. `<name>`, can be used to annotate the data that is contained within the block the brackets form with an intuitive meaning. For example, name `<solution>` is reserved and is used to enclose the final solutions computed for each try.

Results are different from try to try. All data of a block with number `<value>` is associated with the try `<value>` refers to, independently of the location of the block in the output. However, in order to better separate the try independent data from the try dependent data, blocks containing try dependent data should be located within the reserved block indicated by brackets `begin problem <name>` and `end problem <name>` intended to concentrate all data related to and thus dependent of the individual tries. The results for each try typically include an indication of the development during runtime of one or more performance measures and perhaps of (partial) candidate solutions. Additionally, final values of performance measures and final solutions are often output. These two kinds of data should be put into two predefined blocks.

Data reflecting the evolution of performance measures or candidate solutions during runtime should be bracketed by `begin try <value>` and `end try <value>`. The number `<value>`, again, indicates the number of the try. The data inside these brackets constituting so called *try blocks* represents a list of results, each entry using one line. Each line can consist of a list of different name value pairs called *fields*. Names, values and fields of

[4]Spaces and tabulators, in this case the newline character ('\n') does not count as whitespace

an entry are all separated by a whitespace. As a consequence, the name and value parts of fields must not contain any whitespace. Ideally, each line contains two field, i.e. name value pairs. The first field consists of the name of one of the performance measure from the module's interface output and its value. Equally, the first field can contain an encoding of a candidate solution (without using whitespaces, though) or any other kind of relevant information. The second field then contains the point in time during runtime in any time scale (seconds, steps, cycles, and so on) when the measurement or rather output of the information took place. This field could be named time. The name of the first field labels the whole entry. Entries with different labels can interleave arbitrarily. For example, if every new best result of a local search heuristic together with the time of discovery is output, this yields a list of new best results and time pairs that can be used to plot a solution-quality vs. runtime trade-off curve. As another example, whenever a partial solution is updated during runtime, the new partial solution together with the time of change can be output as a new line containing two fields, one for the solution named `solution` and one for the time of change named `time`. The labels of an entry are important. When executing a data extraction script, only entries with known labels will be considered. Entries with unknown labels will be ignored. By default, all entries with labels identical to one of the performance measures of the command line interface definition of the last module of an algorithm are known to the testbed. The set of known labels can be changed, however. See section about writing data extraction scripts 4.3 on page 192 for further information about this topic. Note that no further nesting of blocks inside the `begin try <value>` and `end try <value>` blocks is allowed and that empty lines will simply be ignored when extracting data with the testbed's data extraction language. The latter holds for all types of blocks and the whole output in general.

Data encoding the final results of a try in the form of final performance measure values and an encoding of final solutions can be listed in blocks enclosed by `begin solution <value>` – `end solution <value>`. These blocks called *solution blocks* have to be placed after `end try <value>`. Each line in such a block is viewed as *one* single field, i.e. as a name value pair separated by the first occurring whitespace. Any string before the first whitespace is identified as the name of the field and anything after the first whitespace until the end of the line is taken to be the value of the field. This value can be a numerical value or a string encoding, for example, a solution. Fields named after a performance measure as exported by the last module in its command line interface definition and fields with names `seed` and `solution` for the seed and the final solution are reserved within these solution block and will be detected automatically by the testbed. The field's values can be accessed through a variable with their names (only starting with a trailing `$` as all variables in PHP do. See section 4.3 on page 192 about writing data extraction scripts for more details).

As mentioned before, some information is independent of tries and relates to the run of a job altogether. This information is separated into blocks with reserved bracket names, too. These brackets violate the *generic bracket* format because no additional `<value>` is

needed (since they occur only once in the output). These reserved brackets are described next.

Although the performance measures the last module of an algorithm and hence the algorithm itself produce have to be exported by the command line interface definition of the modules, these performance measures can be stated in the output file, too. When executing a data extraction script, the testbed will automatically provide the performance measures from the command line interface definition (see section 4.3 on page 192 discussing writing data extraction scripts). Additionally, having the same syntax as in the command line interface definition format (section 4.2 on page 181), these, and possibly additional performance measures can be repeated in the output global block bracketed by `begin performance measures` and `end performance measures`. The block formed by these brackets is called *performance measures block*. Both sets of performance measures will be provided automatically by the testbed during execution of a data extraction script. However, consistency is not enforced or even checked by the testbed.

The command line call has to be enclosed in `begin call` and `end call` constituting the *call block*. The contents between these brackets is supposed to be a single string. The parameter settings are enclosed in brackets `begin parameters` and `end parameters` and have to consist of one line per parameter. The block defined by these brackets is called *parameters block*. Each line consists of the parameter name and the value in the form of two strings separated by the first occurring whitespace or '=' sign. The parameter names can be arbitrary and do not have to be identical to the parameter flags used by a module of the algorithm. Additional, 'virtual' or rather derived, parameters can be included, too. Thus, any information globally valid for all tries can be conveyed within these brackets. For example, if the problem instance contains an indication of the cost and an actual encoding of the global optimum in case of a combinatorial optimization problem, this information can be stored in the output by including two lines in the `begin parameters` – `end parameters` block with names `optimum_cost` and `optimum_data`, the first followed by a numerical value representing the solution cost of the global optimum, the last followed by a string encoding the actual global optimum. Any encoding can be used here as long as it is represented as a string; decoding the string is not the testbed's task, it only provides the string.

Additional information in the form of name value pairs for each try can be provided by placing this information in blocks enclosed by brackets in the generic format described before called *generic blocks*. For the contents the same restrictions hold as for the parameters block if the content between the brackets is to be extracted by the data extraction language: each line in a user defined block must consist of at most *one* name value pair, i.e. field. Again, the name and value parts are separated by the first occurring whitespace. In contrast to the predefined blocks described earlier, these blocks are not processed by the testbed automatically when executing a data extraction script. Any processing of user defined blocks must be initiated with the a special command

(see section 4.3 on page 192). Even though the user defined blocks as formed by the generic brackets are associated with a certain try, globally valid data can be put into these blocks, too. Such a block just uses a unique name and a dummy try number and is placed somewhere in the output once, usually not within the `begin problem <name>` and `end problem <name>` brackets. Later processing within the data extraction script simply ignores the dummy try number. For example, the last module can append additional information after the `begin problem <name>`, `end problem <name>` block such as information about the system like operating system, CPU, RAM, paths, timestamps, etc in brackets `begin system 1` and `end system 1`. Any other data not enclosed in any of the predefined or generic brackets will be ignored when using the testbed to extract data from an output-file.

Table 2.4 summarizes all blocks of the standard output format.

Block Type	Opening Brackets	Closing Brackets
Generic	`begin <name> <value>`	`end <name> <value>`
Parameters	`begin parameters`	`end parameters`
Performance Measures	`begin performance measures`	`end performance measures`
Problem	`begin problem <name/value>`	`end problem <name/value>`
Try	`begin try <value>`	`end try <value>`
Solution	`begin solution <value>`	`end solution <value>`

Table 2.4: Summary of the blocks of the standard output format

An example output file heeding the standard output format is presented in table 2.5 on page 30. Names `best` and `jumps` indicate the two performance measures that were recorded. Performance measure `best` might represent the main information, while `jumps` indicate some other information relevant for analyzing the behavior of the algorithm.

The testbed installation contains a dummy module written in C (compare to subsection 3.2.1 on page 74) that implements some basic routines for outputting results according to the standard output format. These functions can be reused by means of copy & paste. Additionally, in the compressed `tar` file `DOC_DIR`*/examples/modules/Interfaces-Tools.tgz*, a class written in C++ named `StandardOutputFormat` (files *StandardOutputFormat.h* and *StandardOutputFormat.cc* is available that implements a methods to easily output results in the proper format. For more information about these classes and the dummy module see the documentation and the comments in the code.

```
begin call
  test -i input.in -o output.out -t 100 -x 20 [...]
end call

begin parameters
  maxTries 5
  maxTime 30
  [...]
  optimum 140000
  CPU_TYPE PentiumIII
  CPU_SPEED 800MHz
end parameters

begin problem sko100f.dat

begin try 1
  best 153302     cycle 2       steps 2       time 0.09      k_var 5
  jumps 1         cycle 2       steps 2       time 0.09
  best 152166     cycle 3       steps 3       time 0.16      k_var 49
  [...]
  jumps 302       cycle 906     steps 789     time 0.11
  best 149416     cycle 907     steps 907     time 12.88     k_var 3
end try 1

begin solution 1
  best 149364
  time 15.17
  steps 995
  seed 12345678
  jumps 303
  solution {1,8,3,9,4,2,6,7,5}{(2,3),(9,8),(1,9)}
end solution 1

begin solutiondata 1
  permutation 1 8 3 9 4 2 6 7 5
  min_jump (2,3)
  median_jump (9,8)
end solutiondata 1

begin further_infos 1
  [...]
end further_infos 1

begin try 2

  [...]

end further_infos 30

end problem sko100f.dat

begin further_global_infos 1
  [...]
end further_global_infos 1

begin system 1
  CPU_TYPE PentiumIII
  CPU_SPEED 800MHz
end system 1
```

Table 2.5: Example module output with proper standard output format

The next section explains how modules which meet the interfaces described in the last sections can be combined to form algorithms.

Specification of Algorithms

As mentioned before an algorithm consists of one or more modules. An algorithm is a sequence of one or more modules (as shown in figure 2.5 and in figure 2.2 on page 12), which has an input, requires parameter settings for the different parameters of the different modules and produces an output. Overall, the algorithm can also be seen as a black box, which takes an input and parameter settings and produces an output. The parameters for the algorithm are the combined parameters of the single parameters of each module of the sequence of modules forming the algorithm.

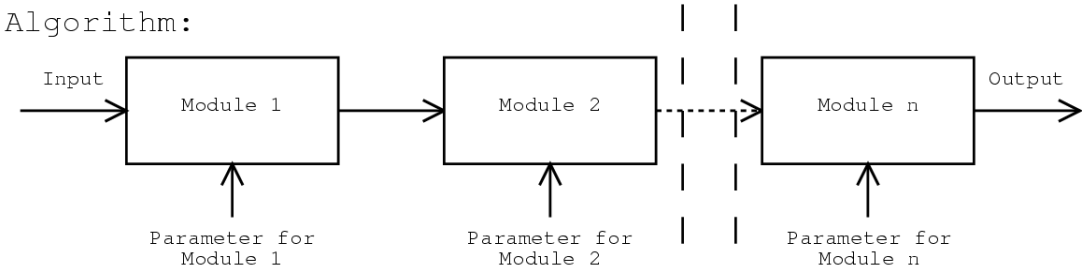


Figure 2.5: Model of an algorithm

The output of each module need not comply to the described output format, except for the last module in the algorithm. The output format of sequence internal modules must only comply to the input format as expected by the next module in the sequence expects. These formats can be specified by the programmer of the modules and the testbed does not need to know anything about them.

Viewed as a black box an algorithm consists of the following components within the testbed:

- A name (ID) of the algorithm, which must be unique,
- a comment,
- a sequence of modules.

The parameters of each module are grouped according to the modules they originated from and provided with some prefix indicating the origin of the parameter. This avoids name conflicts. Some of the parameters can be hidden, i.e. they are excluded from being visible from outside of the algorithm. Default values can be set for the algorithm. *Hidden parameters* or parameters with set default value yield that these parameters can not be set or changed anymore later on.

2.3.2 Problem Instances

All sorts of problem instances for different *problem types* can be stored within the testbed and can also be addressed with a unique ID. Each problem instance must be uniquely be identifiable by its name. A problem instance consists of the following information or rather components:

- A name (ID) of the problem instance, which must be unique within the testbed,
- a comment, describing the problem instance,
- the data of the problem instance (the problem instance itself), no specific format is needed by the testbed and
- additional information like generation time and parameters which describe how the problem instance was generated.

The format of the problem instances does not follow a single scheme, as each problem type comes with its own encoding scheme.

2.3.3 Configuration and Experimentation

Modules, algorithms and problem instances are basic elements of any experimentation effort that are combined with additional information to form configurations, experiments and jobs. The notions of configurations, experiments and jobs from the practice of computational experimentation and its incorporation and representation in the testbed are described next.

Configurations

First, some notions are introduced to clarify some important aspects. A configuration typically denotes a parameter setting of an algorithm, i.e. an assignment of a vector of values to the vector of parameters available or rather visible. Here, this meaning of a configuration is called a *fixed parameter setting*. A configuration in the meaning used in this context is a set of fixed parameter settings . Such a configuration can be specified by providing sets of values for single parameters. Based on these sets, a configuration can be build by construction of a full factorial design[5].

As not all fixed parameter settings of the full factorial design may be needed, it must be possible to remove combinations to form (complicated) subsets of the full factorial design.

[5]All possible combinations of all parameter values for the individual parameters that have sets of feasible values defined (see [17] for further information about experimental design).

Altogether, a configuration consist of

- one algorithm the configuration is based on,
- the name (ID) of the configuration, which must be unique within the testbed,
- a description of the set of fixed parameter settings that form the configuration,
- a comment, describing the configuration.

Experiments

As shown in figure 2.1 on page 7, an experiment is a combination of a nonempty set of configurations in the meaning used here (see previous section) and a nonempty set of problem instances. To identify an experiment the following information is needed:

- The name of the experiment (ID), which must be unique inside the testbed,
- a comment, describing the experiment,
- the set of configurations altogether yielding a set of fixed parameter settings by means of the union operator,
- the set of problem instances, and
- the priority with which the experiment should run inside the testbed or rather with which the jobs generated by the experiment should be run inside the testbed.

Details about jobs are given next.

Jobs

A job results from combining one fixed parameter setting for an designated algorithm and one problem instance. Such a job is considered a single task that has to be executed: Each job has a related algorithm and hence a sequence of modules attached. The task is now to run the algorithm, i.e. the single modules in proper sequence, on the given problem instance and store the output someplace.

The jobs of an experiment are created by considering all possible combinations of fixed parameter settings from the experiment's set of configurations and problem instance from the experiment's set of problem instances. The number of such combinations for an experiment can quickly become huge (e.g. 2 configurations (each configuration consist of 5 different fixed parameter settings) and 5 problem instances , $2 * 5 * 5 = 50$ jobs will be created for such an experiment). It is, however, possible to distribute jobs over computers in a network.

A job consists of the following information:

- The experiment the job belongs to,
- the configuration used to generate the job,
- the fixed parameter setting for the algorithm (which can be determined via the configuration), i.e. the values for the individual parameters together with the parameter flags,
- the problem instance the job runs on,
- the result the job generated,
- the timestamps for job creation, execution and termination,
- the ID of the computer the job was executed on and,
- the status of the job ('executed', 'waited', 'FAILED', and so on).

Derived information for jobs is:

- the algorithm (via the configuration), and continuatively
- the modules of the algorithm.

Notation: The information and objects related to a job as listed just now are denoted as the job's fixed parameter setting, the job's algorithm, the job's problem instance, and so on throughout this document. The output of a job that represents the experimental result is job a job's result or *job result* for short.

As it should be possible to distribute the job over several computers, a *job execution queue* is used to collect all jobs to run. So called *testbed job server* are employed which manage the job execution queue and distribute the jobs over the network of accessible computers. Each computer connected to the testbed server can start a job server and retrieve jobs from the job execution queue and execute (run) that job on the computer. Afterwards, the job is removed from that queue and put in the list of finished jobs.

2.3.4 Statistical Analysis

After experiments have run (or rather the jobs resulting from an experiment), typically a statistical evaluation to examine the results of the experiment is performed. The general goal of empirical experimentation is to gain inside into the laws and regularities that govern observations, in the case of experimentation with algorithms to gain insight into the laws that governs the runs of the algorithms investigated. For example, the experimenter

investigates the influence of parameters on an algorithm's performance. When analyzing the behavior of algorithms, a researcher could in principle do this analysis completely formal and analytically by inspecting the source code. Empirical experimentation of algorithm's behavior is used nevertheless because a formal analysis is prohibitive in many cases, for example because:

The algorithms are too complex or they employ some kind of randomized element. Thus, in order to obtain insight in the behavior of an algorithm, its behavior has to be observed empirically. However, if the behavior of an algorithm is only studied on a limited subset of all possible runs which for obvious reasons is almost always huge, the problem of how to generalize in a sound scientific manner from the observed results to the general behavior of the algorithm arises. In fact, empirical results can be quite misleading. For example, if benchmarks are used for testing, exact postulations can only be formulated with respect to the set of benchmarks used. The same holds for the use of problem instance generators. Generalizations to a bigger set of problems (instances) are subject to immanent uncertainty, since the excerpt of instances used for any experimentation need not be representative of all possible or of all relevant instances. When using randomly generated problems, it could happen, that, by chance, the instances generated are quite easy, suggesting the algorithm tested is very good.

In short, the problem is that even in deterministic algorithms already the unavoidable choice of problem instances used to observe the algorithm's behavior is to some extent random. All these reflections make it imperative to employ statistical analysis in order to generalize results obtained during an experiment to the general case in a sound scientific manner.

In case of the testbed subject of this document, several questions arise:

1. What kind of statistical evaluation is needed?
2. How can the statistical evaluation be integrated in a smoothly and for the user mostly transparent way?

Since usage of an existing statistics package is strongly recommended to compute any statistics needed, the second question splits into several sub problems:

1. Can existing statistics packages be used at all and which?
2. How can these statistics packages be integrated?
    a) How can the data needed for any specific analysis be extracted from the results of a specific set of results?
    b) How can the be data extracted and be conveyed to the integrated statistics package together with the script supposed to analyze the data extracted?

c) How can the statistics package be addressed and controlled?

d) How can the results of the statistical evaluation be displayed by the testbed in turn?

These problems have been solved as follows. The R language [60] is used as the statistical evaluation tool. R provides a huge assortment of statistical methods and, additionally, is a language by its own that can be used to implement individual statistical methods. R can be accessed by external functions. For example, R functions can be called by other programs. In oder to control R, an interface has been defined which enables the testbed to call R in batch mode and execute arbitrary R scripts. The data to analyze is automatically transfered to R by the testbed and the R script is the run on that data. The results returned by R functions are typically of graphical nature or plain text; both types can easily be displayed by any web browser. Extraction of data from sets of job results is achieved by providing a scripting language for data extraction. Note that any other statistics package or other statistics tool could be employed by testbed, too. For this reason, even if only implemented for R yet, scripts for performing a statistical analysis are also called *analysis scripts*. Both notions, R scripts and analysis scripts, are pretty much the same.

The issues of integrating a statistics engine into the testbed as described just now are now discussed in detail. R or analysis scripts are discussed in detail in section 4.4 on page 212. Afterwards, issues related to data extraction from sets of job results are covered.

Statistical Evaluation

The methods of statistical evaluation and analysis of most interest for use with algorithms are exploratory data analysis, hypothesis testing, confidence interval estimation, and model building. Exploratory data analysis is used to scan the results looking for regularities that subsequently can be tested with the hypothesis testing tools or that can be quantitatively bordered with confidence interval estimation. Finally, building a model explaining the general dependencies of the results from the input might be attempted. Typical representatives of these tasks are provided by the R language and thus can be employed by the testbed. Examples are listed next, typical tasks in a statistical data analysis are:

- Drawing plots of
  - solution quality vs runtime trade-off curve (with and without confidence intervals)
  - box-plots

  - runtime distribution (with confidence intervals)

- Computation of statistics of performance measures
  - mean
  - median
  - variance
  - standard deviation
  - minimum
  - maximum
  - quantiles
  - confidence intervals with a specified confidence level

- Statistical model building
  - regression (linear, nonlinear)
  - quality of fit to certain distributions

The following statistical tests are very frequently used for evaluation of results and hence should be provided by the testbed. As R is used to take care are of the statistical part of the testbed all those tests and may more are available in principle. A more detailed description of the information related to statistical tests and statistical evaluation can be found in [14, 11, 12, 13, 15, 16]. Information about practically conducting statistical procedures with statistics packages is provided in [60, 19, 20, 21, 22, 23, 24]:

- Parametric tests:
  - ANOVA in various dimensions
  - goodness-of-fit tests (χ^2 test, Kolmogorov-Smirnov test)
  - t test(normal, paired, two sample)

- Nonparametric tests:
- Wilcox test
- Kruskal-Wallis test
- Other statistical procedures
  - Races ([4], [5])

- Sheffle test
- LSD-tests

All these tests are supported by the testbed by running some R scripts. For different aspects of statistical evaluations, scripts can be written and reused. Such scripts can be used as templates if they are generic. The template R scripts can be filled with the generic information needed and can then be run on the output of the jobs. This avoids copy and paste of scripts, for example if only parameter names are different in the distinct instances of a script. Templates for the statistical evaluation are an essential part of a testbed as already identified in the requirements.

For example, a test of the influence of two parameters on a performance measure is always done the same way. The difference between different evaluations is the name of the parameters to test and the name of performance measure. So a script template for this test can be used by setting variables for the three varying aspects.

To be able to run R, the data needed from the results must be available in a specific format that resembles the table format of relational databases [30, 31, 32, 33, 34, 35]: Each coherent peace of data occupies one line. The columns of the table divide each line in sub components called attributes which hold the atomic peaces of data. For more information about the table format used within the testbed in the context of data extraction is explained in detail in subsection 4.3.1 on page 193. How to extract the data needed for a specific set of (job) results is and how to convert the data into the required format is explained in the next subsection.

Extraction of Data

A data extraction language has been developed to extract data conform to the output format described in subsection 2.3.1 on page 24 for use with the R package. For different extraction tasks and for different format required for the output of the extraction effort, different generic scripts can be written. These templates can be filled with the generic information needed and can then be run on the output of sets of jobs. This avoids copy and paste of scripts, for example when only parameter names are different in the distinct instances of a script.

Extraction scripts are used to extract data from the result of sets of jobs that are conform with the testbed output format as defined in section 2.3.1 on page 24. A set of jobs typically is the result of a query to the testbed database (see subsection 3.5 on page 146). Extraction scripts scan the result of each jobs of a set of jobs, extract certain information and provide them as tables of data in a way similar to tables in relational databases. In this form, the data extracted can easily be conveyed to and processed by the R statistics package.

In order to simplify the construction of extraction scripts a small set of commands and

predefined variables has been developed which automate the most common extraction procedures for job results in the testbed output format. However, when writing extraction scripts the user is not confined to use these predefined commands but can also use any functions of PHP. In general, the extraction script is applied to each job of a set of job result, e.g. the outcome of a search query as described in section 5 on page 33 and any information available for each job (like parameters used , experiment and configuration name etc.) is included in addition to the information extracted by the commands.

The statistical evaluation typically is done in the following way. First, the data is extracted via a data extraction script, afterwards the data extracted is passed with an analysis (R) script to R, which does the statistical analysis or creates the plots and which will finally be transfered back to the testbed for presentation to the user.

2.4 Data Management

In the preceeding subsections a lot of data types or types of objects[6] have been identified that play a major role in experimentation with algorithms. Viewed from a certain perspective the process of experimentation can even be regarded as being primarily a process of dealing with data objects. This perspective further emphasizes the importance and central role of data management support for *any* kind of data related to the process of experimentation. Accordingly, a testbed supposed to automate and help with the process of experimentation must employ some kind of data management system. The field of databases has long been concerned with the topic of organizing and managing huge amounts of data and developed the database management technology [30, 31, 32, 33, 34, 35]. The testbed, too, employs a database to provide a comprehensive data management. Since the data objects identified in the case of experimentation with algorithms all can be viewed as consisting of a set of attribute value pairs it is straightforward to use the long since matured technology of relational databases [36]. The detailed design of the relational database for the testbed, i.e. the structure of interconnections of the single tables storing all data can be found in section 5.1 on page 227. Since the data of the testbed consists of objects which are stored in relational tables of a relational database, each type of object has dedicated at least on table in the database, i.e. tables represent stored data objects.

The usage of a relational database gives access to the full power of this technology, including a powerful query language in the form of SQL [26, 35]. This query language enables searching for and collecting of complex sets of data, especially those sets of data relevant for the statistical analysis, i.e. the sets of job results the statistical analysis is supposed to investigate. The SQL query language, however, might be too complex for beginners, so either an extra search facility with a new user interface has to be developed or the user needs help when formulating queries. The former solution involves the danger that some power is lost, so the second approach was taken: A search query generator was developed.

Search queries can be stored and used later on again and again, as such forming virtual views on the database contents. Stored queries, called *categories*, can be used as filters in submenus displaying the different types of objects and for data extraction. With the help of a query generator the user can interactively refine queries – and learn the SQL query language as a side effect – without losing any power of SQL itself. The search query generator is designed to cover virtually all practical cases. It employs the widely used query by example paradigm [1, 2, 32]. Queries according to this paradigm are formed by filling out blank fields representing the attributes of the objects searched for with the values the objects searched for are required to possess. Further details about search

[6]The notions of data type, object type, kind of object, type of object, type of data are used interchangeably throughout this text.

queries for the objects contained in and managed by the testbed database including most of all jobs and their results, but also algorithms, configurations, experiments, and problem instances are discussed in subsection 3.5.1 on page 146.

2.5 Architecture and Implementation

This section explains in a loose collection of subsections the architectural and implementation specific design and requirements of the testbed.

Since all parts of the testbed use programs that are either licensed by the GNU General Public License (GPL) (*http://www.gnu.org/licenses/licenses.html*) or are completely free, no fees have to be paid to use the full functionality of the testbed (point 9).

2.5.1 Implementation

Based on the requirements listed in section 2.2 on page 11, a framework inspired by phpGroupWare has been developed to fulfill the implementation requirements mentioned there. The phpGroupWare framework is an open source framework which provides services for

1. generating HTML web pages based on templates, and
2. accessing databases or other data sources for managing, retrieving and storing data.

The framework has been developed under the GPL [59].[7] Some parts of the source code of phpGroupWare have been used as a starting point for the testbed framework, mainly to comply with points 4, 5 and 8 of the requirements. Based on the two basic services of the framework for generating web pages and accessing databases, so called applications[8] for managing the different types of objects such as problem instances, algorithms, experiments, and so on, and applications for running experiments, evaluating experiments, and so on have been developed. Together they form the testbed and allow to automate recurring tasks (point 1). It is easy to extend the testbed just by reusing parts or rather services of the testbed framework together with newly developed applications in order to recombine them for new functionality. The way of combining functionality was also inspired by phpGroupWare [44]:

> phpGroupWare is becoming a top intranet/groupware tool and application framework. Written in the PHP programming language, makes it ideal for developers to write add-on applications. PHP is a server side programming language that is simple, cross-platform, and fast.

Accordingly, PHP has been chosen as the implementation language of the testbed, because with the help of PHP it is very easy to write web based user interfaces. PHP is

[7] GNU Public License

[8] Applications denote integral parts of the framework. For more information refer to subsection 5.2.1 on page 232

also a very fast and powerful scripting language. PHP is available on nearly all platforms like Linux, Solaris and other Unix versions and also on Windows. Together with the advantages of the phpGroupWare framework it helps to fulfill point 7 of the testbed requirements of section 2.2 on page 11.

All parts of the testbed are using an object oriented MVC (Model, View, Controller) design pattern [27]. There are classes and objects for representing single testbed components like algorithms, configurations, problem instances and experiments, classes for presenting the data to the user and classes for checking the user input which are also modeling the logic of the relationship between the different components. This directly translates to the different service classes as described in 5.2.1 on page 232. It is possible to use different views, i.e. different methods of presenting the data to the user and different controllers, i.e. different methods of processing the user input. For example, because of the separation of presenting data to the user and the data itself, it is possible to either represent data managed by the testbed to the user via a user interface or to export the data in a machine readable format which can be reread by the same or a different testbed without having to change its internal representation; writing data to a file or to a user interface simply are two modes of export.

Exchange of data has been realized via XML. The XML language [9], a common, human and machine readable format to exchange data has been chosen to enable exchange of data between different installations of the testbed [39, 40]. XML is widely spread for the purpose of exchanging data and consequently helps to fulfill the demands of point 6 of the testbed requirements from section 2.2 on page 11. With the help of XML it is possible to export (and re-import) all single components including their dependencies to other components as shown in figure 2.1 on page 7. All relevant information identified in the previous sections can be exchanged and exported, and hence archived testbed externally, too.

2.5.2 System Requirements and Authentication

The testbed is designed to operate in multi-user mode and to distribute the jobs to be executed over a network of computers. These operations, however, require some functionality provided by the network. These requirements are described next.

In all Linux and Unix systems, one can employ so called *virtual file system trees* (VFS-trees) ([48, 49]). The mechanism enables to include complete file systems of other machines of a network of computers [10] into the filesystem on the local machine at arbitrary so called *mounting points*. The integrated file system will appear as simple directory hierarchy at the mount point which is accessible as a directory, too. It is irrelevant,

[9] eXtented Markup Language

[10] The notions computer and machine are used interchangeably throughout this document and are supposed to denote the same thing.

where this mounted directory hierarchy actually is located, may it be a hard disk, a floppy disk, a CD-ROM and the like. The commands for *mounting* and *unmounting* are `mount` and `umount` The mounting can be done automatically, even according to a specified patterns. This automatic mounting process then is called *automounting*. That way, it is possible to establish complex directory hierarchies and to include the file systems of (all) other machines in a network of computers to each individual computer and hence enable straight forward access to remote file systems.

Another tool called *network file system* (NFS) ([48, 49]) can be used to pretend a remote file system actually is on a local machine. A server exports part of or a whole file system to a special place of its local file system, e.g. */etc/exports/*, and a client can mount these exports. One computer can even be server and client simultaneously. The command for a client to mount such an export is

```
mount -t nfs <servername>:/<path> /<mountpoint>
```

The imported file system now is available on the client under directory /`<mountpoint>`. This way of automounting is quit old and rather specific to Unix and Linux. Ports to Windows operating systems are rather exotic. These parts has some immanent problems with locking of files to handle concurrent access, security, and others. Nowadays, a combination of automounting and NFS is commonly used for convenience reasons, even if this combination does not make the system more stable.

In order to connect Unix/Linux file systems with Windows file systems, a service called *Samba* [43] and an associated protocol named *SMB* have been invented. With the help of this service, within a heterogeneous network of computers of Unix/Linux and Windows machines, directories of Unix/Linux computers can be integrated into Windows machine file systems and Unix/Linux machines can mount Windows file systems in turn. This way of connecting Windows and Unix/Linux machines of a heterogeneous network is quite performing. The command to do so under Linux is

```
mount -t smbfs //<servername>/<share> /<mountpoint>
```

Placeholder `<share>` either is an actual Windows directory on one of its drives, or it is a symbolic directory. Either way, it has to be cleared to be publicly accessible by its owner under Windows. Windows machines then can also connect to `<share>` on Unix/Linux systems under `<mountpoint>`.

By means of the VFS-tree, established in which way whatsoever, it is possible to access a user's home directory from remote machines under identification ~`<username>`, too. However, to do so, each machine needs to know about any user in the networks, so a common identification and login system is required. Such a system then can resolve the network wide valid identification of a user home directory, ~`<username>`, to the machine it really physically is located on. To test this mechanism, command `getent passwd` can be used. This command resides one level above the typical authentication tools or procedures such as `ypcatpasswd` and displays *all* user known to the system: Anyone

who is not listed in the output of this command is not known anywhere in the system. The typical way to provide this common login system is via the yellow pages network information system mechanism ([48, 49]). It distributes so called *maps* over the network in files such as */etc/passwd*, */etc/groups*, and so on and sets up a network wide login procedure.

The yellow pages mechanism is a quite old kind of network common authentication procedure and can be complemented by newer services. Virtually any modern Linux system nowadays authenticates using a mechanism called *pluggable authentication modules* (PAM) (see [41] for more information). This mechanism enables a separation between applications and services, and the actual authentication procedure. Its main advantages are as follows: it is is easy to handle, it is managed centrally, it is build in a modular manner and it is secure. It is available for other Unix systems such as Solaris ([47]), too. However, it is not as common in these systems as it is for Linux systems.

Figure 2.6 depicts the modular design of PAM.

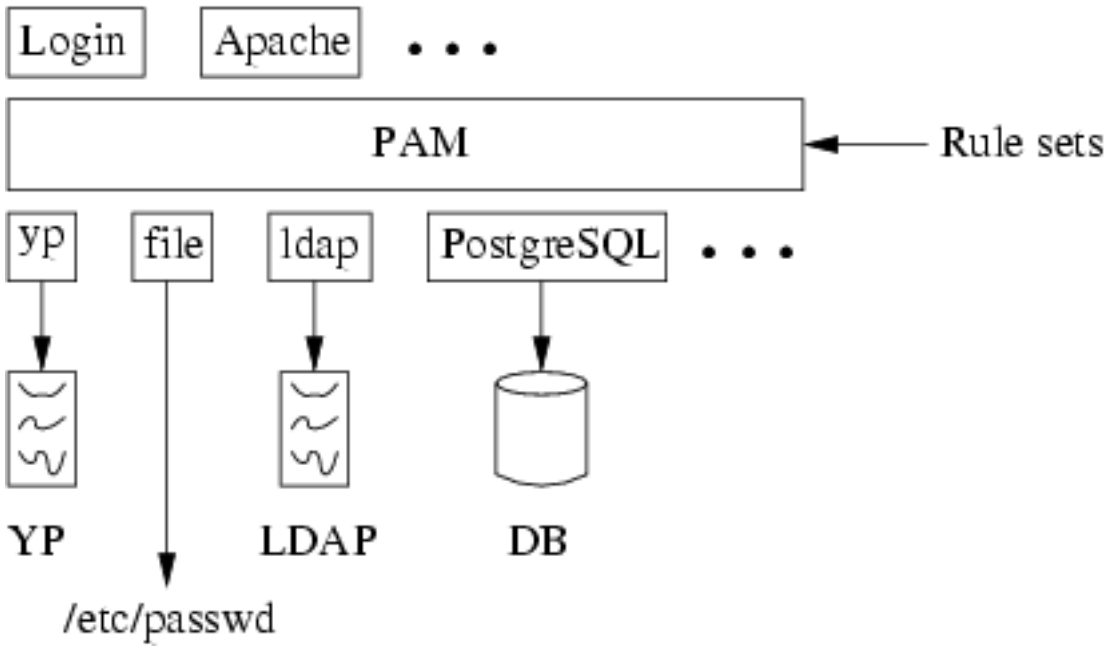


Figure 2.6: Module structure of PAM

PAM serves as a mediator between services requiring authentication such as the login front end, apache, and so on and the actual services that do the authentication such as the yellow pages service, the LDAP, or the PostgreSQL authentication procedure (see figure 2.6). A PAM installation is equipped with a set of rules. Potentially, each client service of PAM requiring authentication can have its own sets of rules how and where to do the actual authentication. If no specific rules are given, general rules will be applied. For example, when an apache web server needs to authenticate a user, it retrieves the username and the password and conveys this information to PAM, which

in turn looks for any rules applicable and according to these applicable rules it relays the actual authentication, for example, to the LDAP ([46]) service.

2.5.3 Database

As mentioned before, a database management systemis used to store any variable data the testbed is concerned with. The database management system used for the current implementation of the testbed is PostgreSQL, because PostgreSQL supports transactions, which are essentially needed when re-importing single components of the testbed. Other databases, which also support transactions, may also be used.

The following nomenclature is used throughout this document. A database management system such as PostgreSQL is also called a database system. Such a system consists of a server also called database server that manages several databases. Each database is what typically is denoted by database, i.e. a collection of tables, in relational format in the case of PostgreSQL. Any single piece of data then is stored in a specific table of a specific database of a database system. Clients can connect to the server and a special database and retrieve, manipulate or store data for this specific database. Access can be restricted to the system as a whole as well as to individual databases.

The typical multi-user installation is done in a local network of computers. Any machine can in principle access any other machine in the network. Since the testbed user interface is web based, once a web server for the testbed is set up, it can be accessed remotely within the network from any machine of the network eligible via any web browser. The remotely connected machines of the network can connect with different user identification each having different access rights to the testbed's database system thus enabling a simple form of multi-user operation. PostgreSQL can manage several databases. Access to each database can be restricted by means of a required login name and password, so only authorized user have access to individual databases. In a multi-user setup of the testbed then, each user typically gets its own database. That way, no user has access to other user's data in their private databases. Each user will have to enter its login name and a password when connecting the first time in a session to the testbed (web) server. The authentication procedure is handled by a web server such as an apache web server [53]. The login information then is used to retrieve more specific information from a configuration file in the user's home directory on the remote client machine that hosts the user's home directory. This file contains information to which database to connect with which password and where to find the module's binary executables and the module definition files (compare to subsection 3.1.4 on page 69). Sharing some databases for a number of user is possible, too. They simple use the same database information in their private configuration files.

Access via browser from the Internet: requires local account on one of the machines in the network

2.5.4 Distribution of Jobs

If the testbed is run in a network of computers that have access to each other via NFS and the mounting mechanism, i.e. each computer can access each other computer via special directories, the execution of jobs can be distributed over the network of computers easily and transparently for the user. This is done by installing a testbed server on one dedicated machine. A testbed server installation consists of a PostgreSQL and a web server and the testbed code itself which is configured to act as a server. On each machine in the network that is supposed to run jobs, the testbed code configured as client gets installed. No PostgreSQL or web server needs to be installed on a client machine. Each user then can enable distribution of its jobs from its database over the machines of the network by starting with its user ID job servers on the client machines. These job servers will read the user specific configuration file, too, retrieving information to which database on which machine/server in the network to connect to with which password. The information are read once when the job server starts and is after that connected to the database the user was working on at starting time as specified in its configuration file *.testbed.conf.php*. If a user changes the database it works on in its configuration file later, any already running job server will be unaffected by this change and will still be connected to old user's database and hence will only execute jobs from this database! When a user has generated jobs, these are put to a virtual queue in its database, also called job execution queue. Each database has associated exactly one such job execution queue. If, for example, several user are working on the same shared database, they share the job execution queue, too. Now, each job server started by any user working on a specific database will continously look for jobs to run in the database's job execution queue. After a job has been run by the job server, is stores the results of the job back to the database it is connected to. Hence, for each database of the testbed's PostgreSQL server, an individual set of job servers has to be run on client machines, since any job sever is only connected to one database. If two user work on the same database and both start individual job servers on the same client machine, these job server will execute simultaneously, perhaps unnecessarily disputing for computation resources. Note that the order in which jobs are executed can not be determined: each job server connected to a database retrieves a new job to execute independently from any other job servers that are connected to this database. Accordingly, it is in principle not possible to predict in advance on exactly which machine a job will actually run. The only exceptions arise through the use of so called aliases.

Additional to the parameterization and the input file, in order to run a job, the job server needs to have access to the binary or rather binaries of a job's algorithm as well as to the wrappers for binaries called module definition files (compare to section 4.2 on page 181). These binaries and module definition files reside in subdirectories of a root binaries directory. The root binaries directory can be specified by each user in its configuration file *.testbed.conf.php* with symbol `TESTBED_BIN_ROOT`. If this symbol is not

defined, `/usr/local/bin/` is used as default. Subdirectory `modules` of the root binaries directory now contains all module definition files, while a binary `XYZ` is contained in subdirectory `XYZ` of subdirectory `<arch>\<os>` of the root binaries directory. The reason to place each binary in architecture and operating system dependent subdirectories (`<arch>` and `<os>`, respectively) is obvious: Each binary has been compiled for a specific target architecture and operating system and will not necessarily work on any other, either. Whenever a job server wants to run a binary, it determines the architecture and operating system it was started on as determined by environment variables `$HOSTTYPE` and `$OSTYPE`, respectively, and looks for the binary in the appropriate subdirectories of the root binaries directory. If no such subdirectories or binary exist, the job server will yield an error message indicating that is was not possible to find the desired binary. If environment variables `$HOSTTYPE` and `$OSTYPE` are not set, default values `default-arch`, and `default-os` will be used. How to set these environmental variables by default for a user or for all users is explained in section 4.6 on page 217.

The computers of the network the testbed is installed on that are running job servers and that connected to the testbed server installation can be categorized according to their hardware equipment into equivalence classes of computer with the same computational power which are called aliases. When an experiment is created, the jobs can be confined to run only on a special hardware class. Thus, it is possible to group all computers of the network with the same computing power together and assign an experiment's job to a specific group of computers with equal computing power ensuring that the results of the experiment are not blurred by different intensities of computation for different jobs. See section 3.1 on page 51 for more information about the installation of the testbed.

2.5.5 Statistical Analysis

The statistical evaluation of experiments (point 3) is achieved by integrating an external statistical program called R [60]. R is a free[11] implementation of the statistical *S/S-PLUS*-language [19, 20, 21, 22, 23, 24]. Most common statistical tests are available through R and do not need to be reimplemented for the testbed. As already mentioned, in order to integrate the R package into the testbed, a job result output format together with an extraction language for this output format has been developed. The testbed will apply an extraction script to the set of job results under investigation. The set of job results will be provided by means of queries to the database (compare to section 3.5 on page 146). The extraction script will extract the needed information from the set of job results and will bring the data extracted in a format similar to that of tables of a relational database (see subsection 4.3.1 on page 193). The data extracted then can either be stored permanently in a file with a format capable of reflecting the relational table structure, e.g. as an comma separated file (CSV), or it will be stored temporarily

[11] GNU Public License

by the testbed in an appropriate format. In the former case, the file can used to feed a statistical evaluation running R externally from the testbed. The latter case is used to transfer the data of data to the statistics package directly. Any applied analysis scripts in this case is run directly from within the testbed. This is possibly, since functions of R can be called externally by PHP. The testbed will simply start an R process, connect to it and execute an analysis script which basically is a script in the R native programming language. The results are conveyed back to the testbed and presented to the user automatically on a web page. By these means, R can be addressed and integrated into the testbed transparently and smoothly.

An more detailed treatment of architectural and implementation issues can be found in chapter 5 on page 227 and more specifically in section 5.2 on page 232.

3 User Interface Description

This chapter is concerned with the installation, configuration and usage of the testbed. It starts with a detailed explanation of how to install the testbed on a Linux or Unix system[48]. Next, an example session of how to carry out an experiment with the testbed is presented. Afterwards, all components of the testbed are explained in detail.

Common notions and abbreviations as used in this document are listed in table 3.1 next. Note that these are placeholder that have to be replaced when actually using them, i.e. during the installation.

CLI	Command line interface. Some parts of the testbed are controlled via a shell like xterm, console, etc.
`WEB_DIR`	This is the directory hosting the web applications on the Linux or Unix system used. For example, the default for a SuSE-Linux system [50] is */usr/local/httpd/htdocs/*, while it is */var/www/* for a Debian system [51].
`TESTBED_ROOT`	This place holder denotes the main path where the testbed is installed. It is `WEB_DIR`/*testbed/*.
`TESTBED_TMP_DIR`	This denotes the path where temporary testbed files will be stored for a user.
`TESTBED_BIN_DIR`	This directory is searched for a user's module binaries and module definition files integrated into the testbed.
`WEB_DOC_DIR`	This is the base directory for documentation of applications in the Linux or Unix system used. By default, this path is */usr/share/doc/packages/* for a SuSE-Linux system and */usr/share/doc/* for a Debian system.
`DOC_DIR`	This denotes the directory with the documentation and examples of the testbed. This normally is `WEB_DOC_DIR`/*testbed/*.
filename	Denotes a file or directory.
`<description>`	This description must be replaced with a value.
`#`	Shell superuser or PostgreSQL superuser prompt.
`testbed=#`	PostgreSQL superuser prompt when connected to database `testbed`.
`<db-XYZ>=#`	PostgreSQL prompt when connected to database `<db-XYZ>`.

Table 3.1: Directory names, naming conventions and abbreviations

3.1 Installation

This section describes how to install and configure the testbed and all software packages it needs for its operation. The description comprises the installation in a network of computers[1] for multi-user mode. If the testbed is to be installed locally on a machine, this can be viewed as special case of a multi-user environment which only consists of one server and one client that are run on the same machine. In almost the same manner, the operation in single-user mode is a special case of a the multi-user mode.

Note that mostly it is first described what has to be installed or configured, next the concrete directions are given including the names of files that have to be adjusted. If during installation errors or problems occur, please refer to the troubleshooting section 4.6. Possibly, this problem has already been addressed there.

The first subsection discusses some system requirements for the testbed and the network infrastructure the testbed presupposes. The next subsection then describes the software requirements for running the testbed, i.e. specifies which software and software packages have to be installed in which version for the testbed to run properly. At the moment, there are no requirements for special hardware. The subsequently following subsection explain in details how to install the testbed and how to configure the testbed.

The testbed currently is shipped in three version:

1. A SuSE rpm-package can be installed on a SuSE-Linux system.

2. A Debian package (ending with *deb*) is provided for installing the testbed on a Debian-Linux system.

3. The source (PHP) code of the testbed contained in a compressed `tar` file[68] is provided for the installation on all other Linux or Unix systems.

The software requirements, and installation and configuration description are essentially the same for all Linux systems. When differences occur, such as different commands or filename, the individual details are labeled **SuSE** and **Debian**, respectively. In the subsection describing the installation, an extra paragraph exists that explains the installation of the testbed on arbitrary Linux systems using the compressed `tar` file. The configuration is the same for all Linux systems. The correct software packages for arbitrary Linux systems have only been listed for SuSE and Debian Linux systems. The names for arbitrary Linux systems have to be looked up in the according documentation of the specific Linux system.

[1] Computers are also called machines in this discourse. Both notions are used interchangeably in this document.

3.1.1 System Requirements

The testbed is designed to operate in a network of computers for a number of different user in multi-user mode. In such an environment there will be one server, called testbed server, which operates a web server and a database server. The testbed database server or database server [2] typically contains one password secured database per user. The testbed server is responsible for the web based user interface of the testbed. It uses a web server such as Apache[53], which therefore is installed on the machine which operates the testbed server itself. This web and testbed server machine, simply called testbed server, typically hosts the database server, currently a PostgreSQL server, too, but it need not. The database server can be on any other machine connected properly to the testbed server machine also. On several other machines of the network, testbed clients can be installed. These clients are pure command line versions of the testbed that are responsible for distributing the execution of jobs over the network of computers in such a way as to be transparently for the user.

Each user typically possesses its own database on the testbed's database server which contains all variable data for all experiments of a user. Additionally, each user possesses its own set of executable binaries implementing modules and module definition files. These are located some place in the user's home directory. Now, when a user connects to the web interface of the testbed on the testbed server machine, some kind of access control and identification of the user is necessary. The identification is done using any kind of web based identification protocol/procedure such as LDAP[3] [46] or others. This section describes the installation of a PAM [41] authentication service. Basically, any identification procedure that is supported by the Apache web server via its modules can be used. For more information about existing Apache identification modules see the Apache documentation [53].

With the help of the identification procedure, the testbed server will know the login name of a user and can use this information to access the user's home directory. The current version of the testbed presupposes that the file systems of any computer in the network the testbed operates in are connected using the Unix/Linux mounting mechanism and NFS. In a Linux network, a computer of the network can export a directory which can be imported by another machine in turn. This exported directory now appears for the importing machine as if it were a directory of its own file system, i.e. is if were located physically on the importing machine. So, when all clients of the testbed server export the directories that contain the user's home directories, they should be accessible by the testbed server the same way as local directories (modulo access rights). That way, any remote computer's file system will look like a typical directory of the server's file system and consequently any user's home directory can be accessed as if it were a directory on the server machine itself.

[2] see section 2.5 on page 42 for a nomenclature for the notions concerned with the topic of databases
[3] Lightweight Directory Access Protocol

The home directory of a user needs to be accessed for two reasons:

1. The testbed server and any client needs information on which machine the testbed database server is located, which database is used by a user and which password has to be used for the database access. By hiding the information about a user's database connection in its home directory together with the required identification procedure of the testbed web server some kind of rudimentary access control is established.

2. The testbed server and its clients eventually must be able to find and retrieve the executable binaries implementing algorithm modules and the corresponding module definition files in order to execute them during the course of the execution of jobs.

All this user specific information is concentrated in a user specific configuration file which is located in a user's home directory: *.testbed.conf.php*

Other means to retrieve user specific information necessary to operate the testbed are conceivable but have not been implemented yet. A more elaborate access control is conceivable, too, but seems to be not really necessary: The testbed presupposes that no user deliberately or maliciously wants to access or even destroy other user data but at the utmost by accident. The access control provided by the testbed therefore simply should be successful in preventing such accidents.

After a user has identified itself to the testbed web server, experiments can be specified and eventually started. In order to execute the jobs, job servers have to be started on each client machine in the network that is intended to participate in the experiment by executing jobs of the experiment. Each such machine needs a testbed client installation which basically means that the command line part of the testbed is installed on it. Such a command line testbed client implements most of all a job server. Each user is responsible to start its own set of job servers on the machines in the network the jobs are to be distributed to. If several user work simultaneously with the testbed in the same network, typically some client machine will have several job server running at the same time. Each such job server belongs to one user and processes this user's jobs. Each job server started will connect to the database that is used at this time as specified in the user configuration file *.testbed.conf.php*. This connection will not change when the user changes its current working database by changing the settings in *.testbed.conf.php*, the formerly started job server still only process jobs from the old database. The reason to employ different job servers for different user is to ensure that each user can run its jobs equitably with the other user. Additionally, each job server or rather testbed client needs to know where to find the module binaries and *.testbed.conf.php* module definition files of the user and how to access the user's (current) database. These information differ from user to user and it seemed to be too much effort to equip the testbed clients with a complicated identification mechanism.

In order to start a job server on a computer in the network, the user has to login on the computer, e.g. using the `ssh` mechanism (see [70, 69]), and then start the job server using the command line interface there (compare to subsection 3.4.4 on page 140). The job server now runs under the ID of the user that has started it and with its rights, knows where to find the user's home directory, and accordingly can retrieve any other information necessary by means of the configuration file. A job server will use this information to connect to the proper database, will next scan the job execution queue of this database and will start and execute the jobs waiting there. After a job has been run, the job server stores the job result (as stored as a file in a temporary subdirectory of directory `TESTBED_TMP_DIR`) back into the database and picks the next job to start it or waits and frequently checks back, whether new jobs have been created and are waiting to be executed. The binaries of a job to run by the job server can be accessed easily (search for in subdirectories of directory `TESTBED_BIN_DIR` by the job server), too, since all machines are mounted mutually such that it looks for the job server as if they are simply located in some directory of the current machines file system. That way, the testbed can distribute the execution of jobs over the computers of a network transparently for the user, since after having started the job servers, a user does not need to bother about them anymore until they are to be killed.

Additional to the parameterization and the input file, in order to run a job, the job server needs to have access to the binary or rather binaries of a job's algorithm as well as to the wrappers for binaries called module definition files (compare to section 4.2 on page 181). These binaries and module definition files reside in subdirectories of a root binaries directory. Subdirectory `modules` of the root binaries directory now contains all module definition files, while a binary `XYZ` is contained in subdirectory `XYZ` of subdirectory `<arch>\<os>` of the root binaries directory. Whenever a job server wants to run a binary, it determines the architecture and operating system it was started on as determined by environment variables `$HOSTTYPE` and `$OSTYPE`, respectively, and looks for the binary in the appropriate subdirectories of the root binaries directory. If no such subdirectories or binary exist, the job server will yield an error message indicating that is was not possible to find the desired binary. For more information about this topic see section 4.6 on page 217 and subsection 2.5.4 on page 47.

Note that on computers operating more than one processor, one job server for each processor can be started.

More information about the presupposed computer network infrastructure is given in section 2.5 on page 42.

3.1.2 Required Software Installation

To be able to operate the testbed, the following software must be installed on the machine which hosts the testbed web server called testbed server:

- Apache 1.3.26 or newer [53]
- PHP 4.1.2 or newer [54]
- GNU R 1.4 or newer [60] (used for the statistical analysis)

The machine hosting the database server (typically the same as the testbed server machine) needs the following software installation:

- PostgreSQL 7.3 or newer [56]

The following specific packages of the software listed just now have to be installed on the testbed or database server machine, respectively. For each software as listed just now, the first set of package or module names refer to a Debian-Linux system, the second set refers to a SuSE-Linux system. For all other Linux systems, the appropriate packages and modules have to be looked up in the according documentation:

- Apache:

  Debian apache

  SuSE apache

- PHP:

  Debian php4, php4-pgsql, php-pear
  (and php4-cgi, if a job server is supposed to run on the server machine, too)

  SuSE mod_php4, mod_php4-core

  All modules must have been compiled with option `--with-psql`!

- R:

  Debian r-base

  SuSE rstatist

- PostgreSQL:

  Debian postgresql

  SuSE postgresql, postgresql-libs, postgresql-server

- Testbed:

  Debian testbed_`<version>`_i386.deb

  SuSE testbed-`<version>`.i386.rpm

The proper installation of the required testbed packages is covered in subsection 3.1.3 on page 61. Note that other Linux or Unix systems might label these packages differently. See the according documentation for more information.

If distribution of jobs over several computers in a network is desired, the following packages are required on any client machine[4], too:

- PHP:

  Debian php4, php4-pgsql, php4-cgi

  SuSE php4, mod_php4-core

  All modules must have been compiled with option `--with-psql`!

- PostgreSQL:

  Debian postgresql-client

  SuSE postgresql-libs

- Testbed:

  Debian testbed-client_`<version>`_i386.deb

  SuSE testbed-client-`<version>`.i386.rpm

How to install and configure the command line packages of the testbed is explained in subsection 3.1.3 on page 61. Note that to be able to use the web based authentication feature, the POSIX-extensions (`--with-posix`) must be compiled in PHP to enable the testbed programs to access the home directory information of authenticated user and to load the user's settings. To be able to run the testbed command line tools, a PHP packages has to be installed which was compiled with option `--with-psql` which typically has been done by default in case of the Debian packages/modules. In case of a Debian system it is recommended to use the non-US version of PHP4.

All these packages are available for free and can be found on any Linux distribution like SuSE[5] [50], Debian [51], Redhat [52], or the like.

In order to turn on the needed PHP support of the Apache web server, the configuration file of the Apache web server must contain the following lines (either enter the lines or uncomment them (remove leading comment indicator `#`):

[4] A client machine is exclusively used for the execution of jobs. It only connects to the database on the testbed database server. Such a client machine does not provide its own web interface.

[5] The PHP packages shipped with SuSE 8.0 are broken and so the command line client of PHP is not working. New packages of PHP for a SuSE 8.0 distribution have been build, so the command line client of PHP is working correctly. They are available via the home page of this testbed [62].

Debian
```
AddType application/x-httpd-php .php
AddType application/x-httpd-php3 .php3
AddType application/x-httpd-php-source .phps
LoadModule php4_module /usr/lib/apache/1.3/libphp4.so
```

SuSE
```
AddType application/x-httpd-php .php
AddType application/x-httpd-php .php3
AddType application/x-httpd-php .php4
AddType application/x-httpd-php-source .phps
```

The Apache configuration file is:

Debian */etc/httpd/httpd.conf* or */etc/apache/httpd.conf*

SuSE */etc/apache/httpd.conf*

Do not forget to restart the Apache server with:

Debian `invoke-rc.d apache restart`

SuSE `rcapache restart`

After the configuration of the Apache server, the PHP modules have to be configured. The memory limit for PHP for the testbed web interface can be changed in the configuration file for PHP. The configuration file that contains the settings for the memory limit for the web interface is:

Debian */etc/php4/apache/php.ini*.

SuSE */etc/php.ini*.

The memory limit setting can be found under item `memory_limit` in each file. A line `memory_limit = 8M` indicates that the maximum amount of memory a PHP script started from the web interface (e.g. a data extraction script), may consume is 8 MB. Note that this applies to any such script. So, if running multiple scripts at once, the actually amount of memory used can be quite huge. The maximum execution time for PHP scripts can be changed in the same file with a line `max_execution_time = 30`, in this case setting the maximum execution time of each script to 30 in seconds. Changing the maximum execution time might be advisable for complicated data extraction scripts processing large amounts of job results. Some further settings must be changed for the testbed to run correctly. PHP options `register_globals` and `magic_quotes_gpc` must be set to `off`. This must be done in files:

Debian */etc/php4/apache/php.ini* and */etc/php4/cgi/php.ini*

SuSE */etc/php.ini*

If the testbed is run with `register_globals = on` it seems to run correctly, but the navigation through the testbed may be broken and it might have some other side-effects which prevents the testbed from running properly. In case of a Debian-Linux system, the following line must be appended to file */etc/php4/cgi/php.ini*:

Debian `extension=pgsql.so`

Note that this is sometimes done automatically upon installation of the PHP or PostgreSQL modules, so please check before appending. If this line is doubly in the configuration file, the following error can occur:

```
PHP Warning: Function registration failed -
                      duplicate name - pg_connect in Unknown on line 0
PHP Warning: Function registration failed -
                     duplicate name - pg_pconnect in Unknown on line 0
 ...
PHP Warning: Function registration failed -
           duplicate name - pg_setclientencoding in Unknown on line 0
PHP Warning: pgsql: Unable to register functions, unable to load in
                                                    Unknown on line 0
<b>Database error:</b> Link-ID == false, connect failed<br>
<b>PostgreSQL Error</b>: 0 ()<br>
<br><b>File:</b> /var/www/testbed/common/inc/class.db.inc.php<br>
<b>Line:</b> 131<p><b>Session halted.</b>
```

More topics related to configuring various aspects of the testbed can be found in section 3.4 on page 140. In particular, the subsection about the job server explains how to change the maximum amount of time a job is allowed to run. After this time the job server that started the job assumes that the job has gone dead and updates the job as having failed in the database (compare to table 3.5 on page 123). The default time limit is 12 hours. If this is too short, for example if some jobs need a execution time longer than that, it has to be changed.

Finally, the database server system has to be configured. Currently, only PostgreSQL is supported and hence is described. Access to the database server can be restricted. This restriction can comprise restrictions with respect to which computers or user in a local network are allowed to access which database of the server and whether they have to authenticate or not. The basic settings are the same for Debian- and SuSE-Linux systems and probably for all other Linux distributions, too. They have to be changed in file

Debian */etc/postgresql/pg_hba.conf*

SuSE *~postgres/data/pg_hba.conf*

The basic settings are depicted next:

```
#TYPE  DATABASE  USER  IP_ADDRESS  MASK             METHOD
local  all       all                               trust
host   all       all   127.0.0.1   255.255.255.255  trust
host   all       all   0.0.0.0     255.255.255.255  reject
```

Each line of the table restricts or allows access to the database server. The first column of the table denotes the destination of access, the next two columns denote which database and which user is concerned, the fourth column indicates the IP address of the machine that is granted connection and access to. Using a mask for IP addresses in the fifth column can affect several machines at once. Finally, the last column specifies the type of access granted and perhaps that an authentication procedure is necessary. The first line allows unrestricted access to any database from the local machine, i.e. from the database server machine. For example, this setting is needed for command `psql` that administers the database system to run correctly. The next line allows access to the database server and any database via a local TCP/IP socket. This kind of access occurs when the testbed or the Apache server are installed on the same machine as the database server (which typically is the case) and want to access the database. By definition, IP address 127.0.0.1 represents the local host `localhost`. Finally, the last line serves as fallback or default setting: Nobody else is granted access to any database and hence to the database system as a whole. If machine in a network of computers is supposed to have unrestricted access to any database, `reject` in the last line has to be changed to `trust`. This, however, grants administration rights to any user on any machine in the local network, so be carefully! Sometimes, column `USER` is not supported (see comments in configuration file). In this case, it simply can be skipped.

The last column called `Method` defines which authentication method is used. Several methods are available. The most relevant ones are listed next together with a short explanation. For more information about user authentication see the chapter about client authentication in the documentation of the PostgreSQL database management system [56].

trust Any connection to the database is allowed without any conditions. Whenever it is possible to connect to the database at all, this can be done as any user without having to supply a password.

reject No connection is possible. Together with the columns indicating the affected IP addresses, this can be used to deny a connection to certain hosts.

password The authentication is based on a password, i.e. access is granted only if the connection can be acknowledged by the proper password. The password is sent in unencrypted form.

crypt One of several possible authentication procedures that are based on an encrypted password. Otherwise, it is the same as `password`.

ident Applying this method, the database tries to obtain the user name of the user that wants to connect to the database from the system. Next, it is checked whether the user is allowed to connect. The database user name can be identical to the system user name or it can be mapped to a database user name by the key word following the `ident` keyword. A mapping `sameuser` is used to indicate identical database and system user names.

pam The authentication is done using the PAM (Pluggable Authentication Modules) service (see next subsection for more information).

The PostgreSQL database server perhaps has to be restarted before new settings take effect. This can be done as follows:

Debian
```
invoke-rc.d postgresql restart
invoke-rc.d apache restart
```

SuSE
```
rcpostgresql  restart
rcapache restart
```

The PostgreSQL database server can be started and stopped with commands:

Debian
```
invoke-rc.d postgresql  stop
invoke-rc.d postgresql  start
invoke-rc.d apache stop
invoke-rc.d apache start
```

SuSE
```
rcpostgressql  stop
rcpostgresql start
rcapache start
rcapache stop
```

Finally, in order to access the current status of the PostgreSQL database server, the following command is useful:

SuSE
```
rcpostgressql  status
rcapache status
```

3.1.3 Installing the Testbed

This section is divided into several paragraphs. The first paragraph describes the installation procedure for a SuSE- and a Debian-Linux system simultaneously based on a system specific testbed installation packages. The second paragraph describes the installation on any other Linux or Unix system without a specific installation package, instead using a `tar` archive of the source code directly. The next paragraph describes the installation of the testbed documentation and examples, while the last paragraph treats updating the testbed.

Installation on a Debian- or SuSE-Linux system

To install the testbed server, the subsequent steps must be carried out as root (`#` indicates the root or PostgreSQL superuser prompt on a shell, `testbed=#` is the prompt for the PostgreSQL superuser 'inside' the database system for database `testbed`, and `<db-XYZ>=#` is the prompt when connected as some other user to database `<db-XYZ>`). Installing a testbed client only requires to carry out step 1. b).

1. Installation of the code:

   a) Installation of the provided testbed installation package file for the server installation:

   Debian `# dpkg -i testbed_<version>_i386.deb`

   SuSE `# rpm -ihv testbed-<version>.i386.rpm`

   b) Installation of the provided testbed installation package file for the client installation:

   Debian `# dpkg -i testbed-client_<version>_i386.deb`

   SuSE `# rpm -ihv testbed-client-<version>.i386.rpm`

2. For the next step, the PostgreSQL database server must be running. PostgreSQL can be started with:

   SuSE `# rcpostgresql start`

   Debian \verbinvoke-rc.d postgres start+

   In case of a Debian system, the postgres server can be configured to always login as PostgreSQL superuser when working with the database system using command `su`

   `# su - postgres`

The following steps then are identical for SuSE- and Debian-Linux systems. Only part `-U postgres` of any PostgreSQL command must be omitted in case the command is executed as user `postgres` under Debian.

Note that the PostgreSQL master or server process must be run with option *-i*. This option can be set in variable `POSTGRES_OPTIONS` in file:

SuSE */etc/rc.config* or */etc/sysconfig.d/postgresql*

Debian In case of Debian, this is not necessary.

3. A new user `<XYZ>` registered to the testbed database system is set up by:

   ```
   # createuser -d -P -q -A -E --host=localhost -U postgres <XYZ>
   ```

   The system will ask to enter an initial password for the new user.

4. A database for user `<XYZ>` in the testbed's database system is created with (`<db-XYZ>` is the database name used for user `<XYZ>`):

   ```
   # createdb -U <XYZ> <db-XYZ>
   ```

   If PostgreSQL requires a password, the password for user `<XYZ>` has to be entered. This holds for any command that includes flags `-U <XYZ>`.

5. The necessary database tables are created with:

   ```
   # psql -U <XYZ> <db-XYZ> < TESTBED_ROOT/database/initdb.sql
   ```

6. If an additional database `<db-XYZ2>` for user `<XYZ>` has to be created, the following commands can be used:

   ```
   # createdb -U <XYZ> <db-XYZ2>
   # psql -U <XYZ> <db-XYZ2> < TESTBED_ROOT/database/initdb.sql
   ```

   Per default, only the owner of a database is granted access to it. All other user that are supposed to access the database, too, must be granted access specifically using the SQL command `ALTER DATABASE` or `GRANT`. See the PostgreSQL documentation [56] for more details.

7. If an additional already existing user `<XYZ2>` should be granted access to a database, e.g. `<db-XYZ>`, the following commands will do:

   ```
   # psql -U postgres <db-XYZ>
   ```

   or

   ```
   # psql -U <XYZ> <db-XYZ>

   <db-XYZ>=# GRANT ALL ON DATABASE <db-XYZ> TO <XYZ2>;
   <db-XYZ>=# \q
   ```

Note: In order to access database `<db-XYZ>`, user `<XYZ2>` now only needs to change the database name in its user specific config file (*./testbed.conf.php*) to `<db-XYZ>`, and adjust the password to the one that is required for accessing database `<db-XYZ>`.

8. Access to the database system is restricted by means of passwords. If no password is given for the PostgreSQL superuser `postgres`, the empty password[6] possibly should be changed (see PostgreSQL documentation [56]).

9. Any password restrictions for a newly created database on the database server will only take effect if file

   Debian */etc/postgresql/pg_hba.conf*

   SuSE *~postgres/data/pg_hba.conf*

   is edited (compare to subsection 3.1.2 on page 54). This file defines which client machines are granted which kind of access with which kind of authentication procedure to the database server and the database itself. If jobs are to be distributed over a network of clients, each client must be listed in this file for some type of access and authentication scheme. In order to manage access to the newly created database `<db-XYZ>` of this installation guide, the following lines (same for both SuSE- and Debian-Linux systems) must be appended to just mentioned access configuration file:

```
#TYPE  DATABASE    USER  IP_ADDRESS  MASK             METHOD
local  <db-XYZ>    all                                trust
host   <db-XYZ>    all   127.0.0.1   255.255.255.255  trust
host   <db-XYZ>    all   130.83.26.0 255.255.255.0    password
```

   Each line specifies one single or a number of machines eligible to access the databases of the database server, in this case database `<db-XYZ>`. The first column indicates the socket used for the access, the second and third columns indicate the database and user the access is granted to, the next column contains the IP-address of the machine in the network that is granted access to, the fifth column defines a network mask and the last column indicates which kind of authentication is used. Line 1 repeats the default settings from subsection 3.1.2 on page 54: Access to specific database `<db-XYZ>` is allowed. This line indeed is redundant to line 1 in the last subsection, but makes it more explicit. The next line is redundant to line 2 from before, too, again to make it specific and explicit for database `<db-XYZ>` The last line, however, allows access to database `<db-XYZ>` from the host with IP `130.83.26.0` of the local network, but only if the user has logged in or rather authenticated itself before. The authentication must follow the `password` restrictions (see subsection 3.1.2 on page 54. In case the web interface is not installed

[6] The password for the PostgreSQL superuser `postgres` sometimes is set to be 'postgres'.

on the same machine as the database server, this authentication is done via the Apache server when the web interface of the testbed accesses the database. Additionally, the required authentication procedure (here: `password`) is used by any command line client on a client machine, for example when running a job server. The job server clearly needs to access a database of the testbed database server. The login information in this case stems from the user information of the client machine on which the job server was started on. The IP-addresses may differ from network installation to network installation. To find out the correct settings for a specific local network please contact the local network administrator. With such a configuration it is not necessary to provide a password to access the database on the local computer (first line). If a remote connection is established via network to this database, the connection must be authenticated with the password given to the additional user. Sometimes, column `USER` is not supported (see comments in configuration file). In this case, it simply can be skipped. A setting `password` will forward the authentication the system to the procedure as described next; an individual and explicit authentication of a user on side of the database system is not possible and superfluous anyway then.

File `TESTBED_ROOT`*/config.php* contains the system wide default settings for the testbed such as the default database to connect to. After adjusting the settings in array `$GLOBALS['dbconfig']` for database `<db-XYZ>` established before (see 3.1.4 on page 69) the testbed will be available via address

`http://<servername>/testbed/index.php`

or

`http://<server-IP-address>/testbed/index.php`.

If `password` in the last line changed to `trust`, no authentication is necessary in order to access database `<db-XYZ>`. In this case, anybody from the local network can access database `<db-XYZ>`. Note that this access is unrestricted and equipped with administration rights, so be carefully! Using method `password` does only work in conjunction with carrying out the subsequent two steps. Without authentication turned on, however, they can be omitted. In this case, one global database is used by any testbed user. Since no authentication is possible, different user can not be distinguished. In detail, a variable `$_SERVER["REMOTE_USER"]`which is used by the testbed configuration file `config.php` is not set and hence default settings are used for all user. The default settings, including the name and access information for the globally used database are set in file `config.php`. These settings can be changed to connect to another globally valid database. For more information about the configuration file, refer to subsection 3.1.4 on page 69.

10. Next, if required, authentication has to be set up and turned on. In order to do so, first the following lines must be added to file

Debian */etc/apache/httpd.conf*

```
<Directory "/var/www/testbed">
AllowOverride AuthConfig
</Directory>
```

SuSE */etc/httpd/httpd.conf*

```
<Directory "/usr/local/httpd/htdocs/testbed">
AllowOverride AuthConfig
</Directory>
```

Now, a file *.htaccess* can be provided in directory `TESTBED_ROOT`. This file determines how authentication via web access is carried out on side of the Apache server. This file has to contain the following basic settings for an authentication scheme:

```
AuthType Basic
AuthName "Testbed User login"
require user <authorized-user1> <authorized-user2> ... <authorized-userN>
```

where `<authorized-userX>` is the login name of the network the testbed runs in for a user that is supposed to access the testbed. Any user that needs access has to be entered there with its login name. Note that the login name entered here are not the user names for the databases of the testbed!

The following instructions explain how to set up a PAM (Pluggable Authentication Modules) service which is the standard authentication environment for any Linux system and most modern Unix systems (compare to subsection 2.5.2 on page 43). Other authentication services such as LDAP ([46]) can be used directly as well. In this case, the specific documentation has to be consulted for how to do this.

11. PAM

    First, the apache module for PAM has to be installed. This package is called

    Debian libapache-mod-auth-pam

    SuSE apache-contrib

    Next, in file

    Debian */etc/apache/httpd.conf*

    SuSE */etc/httpd/httpd.conf*

    the comment character (`#`) at the beginning of line

    ```
    # LoadModule pam_auth_module /usr/lib/apache/1.3/mod_auth_pam.so
    ```

    has to be removed.

Testing whether authentication works is easy. First, when the authentication is properly turned on by the Apache web server, a variable `$_SERVER["REMOTE_USER"]` is available in PHP.

This variable contains the login name of the user that has just authenticated itself.

To view the contents, set up a file *info.php* with contents

```
<?php phpinfo();?>
```

in `TESTBED_ROOT` and open it via the web server with a browser (directly opening it will not trigger any authentication, since this would be a local access and the PHP contents simply would not be interpreted and hence the commands would not be carried out). Next, one should be prompted for a user name and a password. After entering them, the variable should be displayed at some place of the upcoming web page. This page simply displays all system information variables available in PHP.

Accessing the testbed under its address `http://<servername>/testbed/index.php` will then also pop up a user input request for entering the username and a password.

Without changes, the authentication rules used are the standard PAM rules. If they are to be changed, it can be done in file

Debian */etc/pam.d/httpd*

SuSE */etc/pam.d/httpd*

See [41] for more information about how to do this.

Other Linux and Unix systems

The following installation instructions describe the installation of the testbed server for an arbitrary Linux or Unix system using the PHP source code of the testbed in the form of a compressed `tar` file directly. In particular, the `WEB_DIR` directory might differ. Installing a testbed client only requires to carry out step 1 and 2. The `tar` files are the same for the server and the client version.

To install the testbed the subsequent steps must be carried out as root (`#` again is the root prompt):

1. Installation of the provided testbed PHP source code from the compressed `tar` file with:

   ```
   # cd WEB_DIR
   # tar -xzf testbed-<version>.i386.tgz
   # cd testbed
   ```

2. Next, two shell scripts for using the command line interface of the testbed have to created in `TESTBED_ROOT`*/bin/* and next copied to the proper binary directory of the system. The first shell script is named *testbed* and has to have the following contents (replace place holders first):

```
#!/bin/bash
php -C -q -d memory_limit=128M TESTBED_ROOT/bin/cmd.php "$@"
```

Note that command `php` might be called `php4` or else on other Linux systems. Perhaps `php4` has to be linked to `php` or the command has to be changed in the script.

The second shell script named *testbed-check* should contain (replace place holders first):

```
#!/bin/bash
php -C -d memory_limit=128M TESTBED_ROOT/check.php "$@"
```

These two files now have to be made executable and can then be copied to a directory in the path so they are executable from anywhere in the system:

```
# cd TESTBED_ROOT/bin
# chmod a+x testbed
# chmod a+x testbed-check
# cp testbed /usr/local/bin/
# cp testbed-check /usr/local/bin/
```

Directory */usr/local/bin/* typically is the directory that contains all executables of the system. It might also be */usr/local/* on some systems.

3. The set up of the database server and the arrangement of individual databases ((steps 2 – 7) and user works according to the way it is done for SuSE- or Debian-Linux systems. The appropriate commands for the specific Linux- or Unix system have to be looked up in the PostgreSQL documentation specific to the specific system.

4. The database password restrictions have to set up according to the setup for SuSE- or Debian-Linux systems (steps 8 – 11). Any differences, e.g. for the authentication file have to be looked up in the corresponding documentation. Any PHP options and the Apache web server access control settings might have to be changed, too.

Installing the Testbed Documentation

The documentation of the testbed contains this user manual in various formats as well as several generic and example data analysis and extraction scripts. All examples from section 3.2 on page 74 are provided, too. Additionally, some C and C++-code is provided, e.g. a example dummy module and classes implementing proper parsing of command line parameters and output functionality conform to the standard output format.

In case of SuSE- or Debian-Linux systems, the package containing the documentation files simply has to be installed with:

Debian `# dpkg -i testbed-doc_<version>_i386.deb`

SuSE `# rpm -ihv testbed-doc-<version>.i386.rpm`

For other Linux and Unix systems, the compressed `tar` file containing the example and documentation files has to extracted in the documentation directory:

```
# cd DOC_DIR
# mkdir testbed
# cd testbed
# tar -xzf testbed-doc-<version>.i386.tgz
```

Updating the Testbed

Updating a testbed installation is easy. In case of SuSE- or Debian-Linux systems only new installation packages have to be installed in update mode:

Debian
```
# dpkg -u testbed_<version>_i386.deb
# dpkg -u testbed-client_<version>_i386.deb
# dpkg -u testbed-doc_<version>_i386.deb
```

SuSE
```
# rpm -Uhv testbed-<version>.i386.rpm
# rpm -Uhv testbed-client-<version>.i386.rpm
# rpm -Uhv testbed-doc-<version>.i386.rpm
```

In case of other Linux system installations, the testbed installation procedure concerning the compressed `tar` file and possibly the creation of the additional utility files has to be repeated, thus overwriting the old code.

If a new documentation package has been installed or updated, the new versions of the user manual in PDF and HMTL format have to be copied to the place where the online help of the testbed (submenu 'User Manual') can find it:

```
# cp DOC_DIR/usermanual.pdf TESTBED\_ROOT/manual/EN
# cp -a DOC_DIR/html TESTBED\_ROOT/manual/EN
```

See the testbed home page for any version specific further update tasks [62].

3.1.4 Configuring the Testbed

After the installation of the required software and the configuration of the system software, the testbed itself finally has to be configured, too. This is mainly done by creating user configurations in the form of user specific configuration files. This section regards any Linux or Unix system the testbed is installed on. It provides information about how to configure the testbed globally and individually for each user. The configurations tasks discussed are concerned with specifying database connections, specifying user specific directories containing module binaries and module definition files, and some adjustments of appearance of the testbed's web based user interface.

The settings described here can be made in either the global server side configuration file `TESTBED_ROOT`*/config.php* or in each user specific configuration file *~/.testbed.conf.php* (~ indicates the user's home directory). Any user specific setting will overwrite any corresponding global setting. Accordingly, if no user specific settings were made for some items, the according global settings will take effect.

In order to operate the testbed in a distributed and/or multi-user environment, it must be possible to share the files of each user over all computers in a network, e.g. via NFS[7] or SAMBA[8] entailing that all files can be found at the same place in the file system (see subsection 2.5.2 on page 43).

Global configuration file `TESTBED_ROOT`*/config.php* must be checked to ensure that the settings correspond to the system on which the testbed was installed. Whether all settings for the testbed are correct can be checked on page `http://localhost/testbed/check.php`. This page also shows the current status of the testbed and the last errors or problems that occurred and can be accessed via submenu 'Testbed Status' (see also section 3.3.13 on page 133). For checking the settings of a client installation, issue `testbed-check` on the command line interface.

In order to make the testbed work for a user (account), each user needs a user specific configuration file with name *.testbed.conf.php* in its home directory. This file should contain the following settings:

```
<?php
define('TESTBED_TMP_DIR', '/tmp/<username>-testbed');
define('TESTBED_BIN_ROOT', '<user home dir>/testbedbin');
```

[7] Network file system
[8] A Windows SMB(Server Message Block)/CIFS(Common Internet File System) file-server for UNIX

Directory/path `TESTBED_TMP_DIR` describes where temporary data (e.g. the output of jobs, plot output of statistical analysis, etc.) will be stored. Any data stored there will be deleted by the testbed automatically as soon as it is not needed anymore. `TESTBED_BIN_ROOT` is searched for recursively for any module binaries of the user and the corresponding module definition files.

Additional to the directory definitions in file, the user must specify the information for connecting to the testbed database server and the specific database the user wants to use in file *~/.testbed.conf.php*. The following settings must be made:

```
$GLOBALS['dbconfig'] = array (
  'db_host' => 'Name or IP-address of host running the database server',
  'db_name' => 'Name of database',
  'db_user' => 'User name for database access',
  'db_pass' => 'Password for user and database access',
  'db_type' => 'pgsql',
);
```

The settings `Name or IP-address of host running the database server`, `Name of database`, `User name for database access`, and `Password for user and database access` specify on which computer the testbed database server runs, which database the user wishes to connect to, under which user name the user can access the database, and which password is required for that database access, respectively. The last entry must be changed, if another type of database management system instead of PostgreSQL is used. Currently, no other database management systems are supported by the testbed so this entry remains unchanged.

The global configuration file `TESTBED_ROOT`*/config.php* contains as default setting the following database access configuration entry:

```
$GLOBALS['dbconfig'] = array (
  'db_host' => 'localhost',
  'db_name' => 'testbed',
  'db_user' => 'testbed',
  'db_pass' => 'testbed',
  'db_type' => 'pgsql',
);
```

If any user has its own valid for all user settings, these need not to be changed. Otherwise, they have to set to sensible values. For example, if the testbed is to be operated on a single machine with only one user or each user using the same database named `testbed` with password `testbed` under the database user name `testbed`, the example settings should work fine.

Possibly a user wishes to maintain different databases for different kinds of experiment the user conducts. To do so, the different databases have to be created such that the

user is eligible to access them as described in subsection 3.1.3 on page 61. Whenever the user then wishes to switch the current database in use, he or she simply changes the settings in the `$GLOBALS['dbconfig']` construct to the appropriate settings for the new database manually. In order to simplify this switch, different such constructs with different settings can be listed in the user specific configuration file and only one is not commented out (PHP comment brackets are`/*` and `*/`).

Each user can make some further small adjustments on the web based user interface of the testbed (compare to subsection 3.3.12 on page 132). These preferences can be set in the user specific configuration file, too, with lines listed below. They define the settings for the number of entries that are displayed per page in a submenu, the number of rows a larger text input field should have, and how many columns a text input field should have, respectively. In order to set these preferences to values `10`, `20`, and `70`, respectively, add the following lines to the configuration file:

```
$GLOBALS['user']['preferences']['common']['maxmatchs'] = 10;
$GLOBALS['user']['preferences']['common']['textrows'] = 20;
$GLOBALS['user']['preferences']['common']['textcols'] = 70;
```

These settings are contained in the global testbed configuration file, too, and hence can be changed there, too.

Demo Mode

The testbed can be configured to run in a demonstration or simply demo mode. The demo mode features some restrictions which make it possible to safely install the testbed accessible via the Internet for anyone even without explicit registration and identification. In particular, the number of different objects that can be stored in the database such as problem instances, algorithms, configurations, experiments, jobs, and scripts can be confined. This way, the database can not exceed a certain size and hence the disk space of the machine the testbed database server in demo mode is installed on can not run out. Otherwise, malicious and massive insertion or creation of data in the testbed database by means of the testbed web interface could finally occupy all disk space of the database server machine and hence effectively shut down the testbed operation.

The maximum number of different types of objects allowed in the database in demo mode can be adjusted in the global configuration file `TESTBED_ROOT`*/config.php*. First, the demo mode has to be turned on. This can be done by uncommenting the following line:

```
define('DEMO_MODE','YES');
```

If and only if variable `DEMO-MODE` is defined to be `YES`, the testbed will run in demo mode. Next, the individual limits with respect to a maximum number of allowed objects in the database can be adjusted by changing the according numbers for the different object

types in the following array (`resultscripts` refer to data extraction script, `rscripts` refer to analysis (or R) scripts):

```
$GLOBALS['max_elements'] =  array (
 'problemtypes' => '10',
 'probleminstances' => '100',
 'modules' => '5',
 'algorithms' => '10',
 'configurations' => '50',
 'experiments' => '100',
 'jobs' => '1000',
 'resultscripts' =>  '50',
 'rscripts' => '50',
 'categories' => '100'.
 );
```

No problem instances or jobs can be uploaded, neither directly by creating a new one nor by importing one via XML (compare to subsection 3.3.4 on page 104). With the help of these restrictions, no huge data can be inserted into the database of a testbed in demo mode operation. In always the same manner, any jobs run by the demo mode testbed must not produce too huge result files when configured disastrously: even if the number of jobs is restricted, jobs in principle can produce enormous result files. This danger can be remedied by installing only proper modules for the demo version; user connecting from the Internet typically do not have access rights for the demo mode testbed server machine or any other client in the local network connected to this server and hence can not integrate their own modules, since this can only be done via the command line interface (for exactly this reason, compare to subsection 3.2.2 on page 76 and section 4.2 on page 181).

Hint for even more security: The kernel of a Linux/Unix system can be configured to limit the maximum amount of main memory, CPU time, and/or disk space any process might use. The command to do so is `ulimit`. See the manpage [73] for more information.

Altogether, running a testbed in demo mode should be save enough to allow arbitrary user to access it via Internet and to perform test experiments on the modules and problem instances provided by the demonstration installation.

The testbed example dummy module (see 3.2.1 on page 74) can be compiled to run in a demo mode. This demo mode version of the example module can only be configured in such a way that the result files produced are relatively small (< 250KB). To compile the demo mode version simply issue `make Dummy-Demo-Mode` in the directory the example dummy module source code is located (which is `DOC_DIR`*/example_session/source_code*). The resulting executable is named `Dummy-Demo-Mode`. The source code of the example dummy module is also contained in directory `DOC_DIR`*/examples/modules* as compressed `zip`-file named *Testbed-Dummy-Module.zip*. A fitting module definition file is located

in the same directory and is named *module.Dummy-Demo-Mode.inc.php* or it can be created automatically (see section 4.2 on page 181). The demo version of the example dummy module differs in that the ranges of certain crucial parameter values are strictly constrained. That way, when it is configured, any configuration and execution of the demo mode dummy module will produce result files that do not exceed a certain size. In particular, parameters `tries`, and `maxMeasures` must not exceed a certain value, otherwise they get pruned to the maximum allowed value, while parameter `finallyWait` can not be configured anymore and is always set to `false`.

3.2 Getting Started

This section explains with the help of an example session, how to specify, start and evaluate a simple experiment. The module used for this example session is discussed briefly in the next subsection before starting the example session.

3.2.1 Example Module

This is a brief explanation of an example module shipped with the testbed. The example module often is also called *dummy module* or *dummy* in short This module can be created by calling `make` on the `CLI` in directory `DOC_DIR`*/example_session/source_code*. It is intended for testing purposes for the testbed. The module simulates the behavior of a Metaheuristic used to solve combinatorial optimization problems by producing output similar to what would be produced by the Metaheuristic. The dummy also gives an example for a proper application of the command line interface definition format and the standard output format.

The dummy provides five performance measures `best` and `worst`, `steps`, `stepsBest` and `stepsWorst`. The output contains for each of the two performance measures `best` and `worst` data describing the evolution the two performance measures. Each time a new `best` or `worst` performance measure is found by a Metaheuristic it is output in a new line (compare with the discussion of the standard output format 24). The dummy simulates this output. All measurements together can be used to form a solution quality trade-off curve. Additional performance measures are `steps`, `stepsBest`, and `stepsWorst`. They represent the number of times a any performance measure, performance measure `best`, and performance measure `worst` have been output, respectively.

The entries forming the trade-off curves are simulated by using a number of time points between a maximum and a minimum time, both being greater than zero. For each time point, a virtual value of each performance measure is computed with the help of a function that reflects the typical appearance of of trade-off curve as encountered in combinatorial optimization. Both trade-off curves for performance measures `best` and `worst` essentially are identical; they are simply mirrored at the middle of the time range.

Via parameter settings the time range, the maximum number of data points for each performance measure, the function employed to compute the virtual performance measure values, a degree and type of randomization for time points of virtual measurements, the degree and type of randomization of the values of the performance measures taken, a range for the performance measure values, the number of tries, the seed for the random number generator, and the input and output files according to the command line interface definition format of the testbed can be controlled. The performance measures are computed by first distributing the number of time points equally over each tenth power in the specified range of time points. Next, for each such actual time point the virtual

performance measure value is computed using the requested function. Let `minTime` and `maxTime` be the range borders for time points, `maxMeasures` be the maximum number of time points, `randomTime` be the degree of randomization for time points, and `randomY` be the degree of randomization for the virtual performance measure values at the time points. First, values $a := \text{floor}(\log_{10}(\texttt{minTime}))$ and $b := \text{ceil}(\log_{10}(\texttt{maxTime}))$ are computed. For each integer in $[a, b-1]$, $c := \text{floor}(\frac{\texttt{maxMeasures}}{a-b})$ time points are distributed equally over $[10^a, 10^{a+1}]$. For each such time point the virtual performance measure value is computed. Next, the time points and virtual performance measure values are randomized according to either a uniform distribution over

$[tp_{\,i}$ - `randomTime` $\cdot\ (tp_{\,i}$ - $tp_{\,i-1})$, $tp_{\,i}$ + `randomTime` $\cdot\ (tp_{\,i+1}$ - $tp_{\,i})]$ and

$[tp_{\,i}$ - `randomY` $\cdot\ (tp_{\,i}$ - $tp_{\,i-1})$, $tp_{\,i}$ + `randomY` $\cdot\ (tp_{\,i+1}$ - $tp_{\,i})]$, respectively,

or over a Gaussian distribution with mean defined as $\mu = tp_{\,i}$ and a standard deviation of $\sigma =$ `randomTime` $\cdot\ (tp_{\,i+1}$ - $tp_{\,i})$ and $\sigma =$ `randomY` $\cdot\ (tp_{\,i+1}$ - $tp_{\,i})$, respectively, with $tp_{\,i}$ being the i-th time point. Next, time points and virtual performance measure values are sorted independently, possibly building new time value pairs. All measurements for performance measure `best` are uniformly elevated by a constant above the lower limit `yMin` such that the minimum over all tries is just above the lower limit. Finally, all measurements outside the specified ranges for the time points and performance measure values are discarded.

The input file is used to scan for an integer occurring. The first substring construable as an integer is taken to be the instance size. Other information is not needed and will be ignored.

Currently, three function for producing the simulated trade-off curves are available with yMin being the minimum for values of performance measures as set by the parameters:

1. $f(x) = \frac{1}{x}$ (+ yMin)
2. $f(x) = \frac{1}{x^2}$ (+ yMin)
3. $f(x) = \frac{1}{ln(x)}$ (+ yMin)

If parameter `finallyFail` is set to true (default is false), the program will exit with an exit code unequal to zero, indicating an error to the caller. This is useful with respect to the testbed to test what happens if a program fails. The output and the computation remains unaffected.

If parameter `finallyWait` is set to true (default is false), the program will wait for additional maxTime seconds using system call sleep at the end of execution. The output and the computation remains unaffected. To indicate progress, dot will be printed each couple of seconds. Note that the time for computing adds to the total execution time! Note also that library unistd.h is needed for system call sleep!

Parameter `maxTime` can be set using an arithmetic expression over variable 'n' representing the instance size. The following function in addition to the standard arithmetics (+, -, *, /, %, ^) are available: sqrt, abs, log (natural logarithm), $\log_{10}$, exp, sinh, cosh, tanh, sin, cos, tan, floor, ceil. Note that if the bash indicates an error with this arithmetic expression, simply enclose it in double quotes "'".

For more information about parameters and eligible values refer to the command line interface definition of the dummy by calling the executable with the `--help` option.

The solution encoding for a run of the dummy will be a lot of `performanceMeasure = value` pairs, bracketed by `{` and `}`, separated by commas containing no whitespace.

To compile/create the executable, issue `make` in the directory with the source-code of the dummy. The source code of the exampled module is also contained in directory `DOC_DIR`*/examples/modules* as compressed `zip`-file named *Testbed-Dummy-Module.zip*.

The functions of the testbed dummy written in C can be reused by means of copy & paste. Additionally, in directory `DOC_DIR`*examples/modules*, a compressed `tar` file (`DOC_DIR`*/examples/modules/Interfaces-Tools.tgz*) contains classes written in C++ that implement basic functionality with respect to parsing the command line parameters of a program call and outputting results in proper standard output format. The main classes for parsing parameters are named `Parameter` and `ProgramParameters` (files *Parameter.h*, *Parameter.cc*, *ProgramParameters.h*, and *ProgramParameters.cc*). They implement a convenient specification and parsing method for the command line interface of programs according to the command line definition format. These can be reused, too. Class `StandardOutputFormat` in the compressed `tar` file `DOC_DIR`*/examples/modules/Interfaces-Tools.tgz* (files *StandardOutputFormat.c* and *StandardOutputFormat.h* implements a convenient methods to output results in proper format according to the standard output format of the testbed (see paragraph 2.3.1 on page 24 of subsection 2.3.1 on page 14). File *Interfaces-Tools-Example.cc* implements a demonstration of how to use the interface tools. It can be compiled using command `make`. All other files in the compressed `tar` file are auxiliary classes or files such as *PerformanceMeasure.h* and *PerformanceMeasure.cc* implementing a class to represent performance measures, *RandomNumberGenerator.h* and *RandomNumberGenerator.cc* implementing a random number generator, and *Timer.h* and *Timer.cc* implementing timing functionality. All files are documented using the format of the Doxygen documentation system [37].

3.2.2 Installing a Module

In order to make a module run in the testbed, the first step is to register the module to the testbed. Two files are needed to make the module run within the testbed. The first file is the binary executable implementing the module. The second file needed is the file for registering the module to the testbed called module definition file (see section 4.2 on page 181 for more information about module definition files and subsec-

tion 4.2.1 on page 181 for more information about tool automatically generating module definition files). If the module is compliant to the command line interface definition format, the second file can be generated automatically by calling `TESTBED_ROOT`*/devel/gen_module_from_mhs.php* or `testbed modules makeConform` in the directory with the name of the executable as first argument (perhaps with preceeding `./`) in the directory the binary is located. The user next is required to enter a name for the module as used in the testbed, a problem type the module is supposed to work on, a description and some internal parameters. The input for the internal parameters should be `--finallyWait 0 --finallyFail 0`. Instead of generating a module definition file anew for the dummy module, one can also use the one named *module.Dummy.inc.php* in directory `DOC_DIR`*/example_session/module_definition_file*. This module definition file works fine with the dummy module named *Dummy* in the same directory. The registration of the module is continued via the CLI by (if the directories don't exists in the following, they must be created):

1. Copying the module definition file to the `TESTBED_BIN_ROOT`*/modules/* directory,
2. copying the executable to the directory `TESTBED_BIN_ROOT`*/<arch>/<os>/<modulename>/* (in this case `TESTBED_BIN_ROOT/i386/linux/Dummy/`), and
3. registering the module in the testbed with CLI command

   ```
   testbed module register Dummy
   ```

When generating a module definition file automatically, the appropriate commands with the correct locations will be printed on the console so the use can copy and paste them. After issuing the last command, there should be a message that the registration of the module was successful. The module name is case-sensitive. Only characters 'a' - 'z', 'A' - 'Z', and '0' - '9' are allowed in the module name. All other characters will be removed silently.

3.2.3 Importing Problem Instances

The next step is to import some problem instances for later use in the experiment. This is also done via the CLI of the testbed with command

```
testbed probleminstance add Dummy *.dat
```

All *.dat* files in the current directory are added to the database for the problem type 'Dummy'. Instead of **.dat*, any list of files to be imported can be included in the command. In the example session directory `DOC_DIR`*/example_session/probleminstances* 15 problem instance files named *100-1.dat*, ..., *100-15.dat* are stored and can be integrated. If problem type 'Dummy' does not exists in the testbed, it will be created automatically. More information about managing problem instances in the testbed can be found

in subsection 3.3.4 on page 104. Information about managing problem types is given in subsection 3.3.3 on page 103.

Figure 3.1: Main menu of testbed

3.2.4 Creating an Algorithm

After registering the module and importing some problem instances via the CLI, the next steps are done using the web front end, i.e. the standard user interface of the testbed. The web front end can be accessed on the computer which acts as server. The URL for accessing the testbed locally on a machine typically is *http://localhost/testbed/*. If accessing the testbed on a remote server, 'localhost' has to be replaced by the name (and site) of the remote machine.

The main menu of the testbed is placed permanently on the left side of all testbed pages (see figure 3.1). Each step of this example session can be accessed through the corresponding link in the main menu.

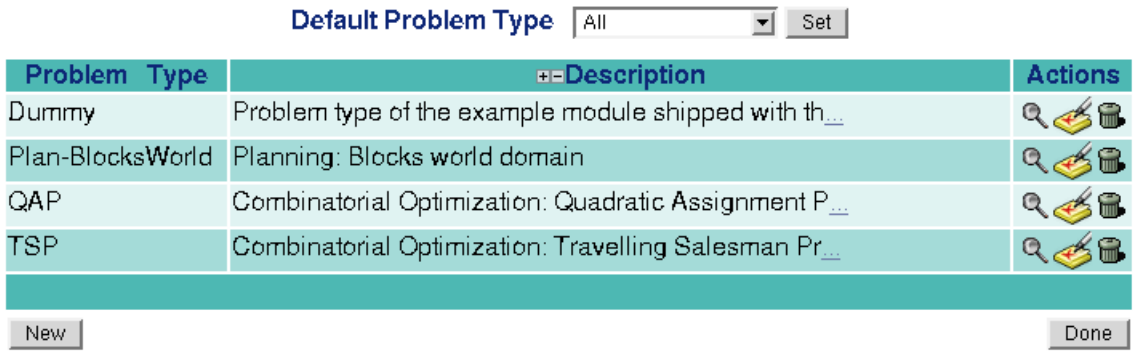


Figure 3.2: Selecting a problem type

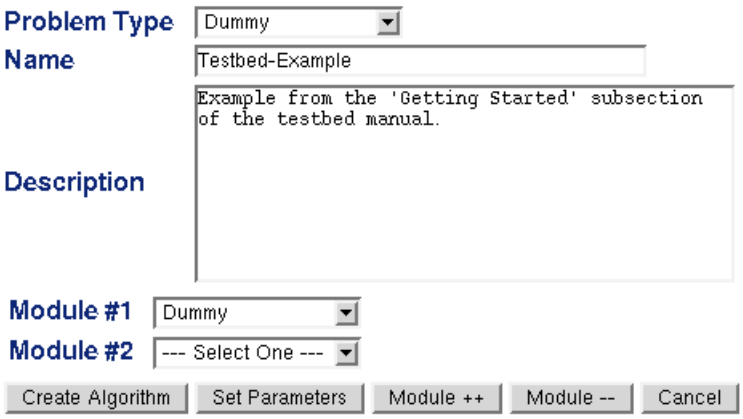


Figure 3.3: Creating an algorithm

The first step is to select a default problem type (see figure 3.2). The advantage of selecting a default problem type is that for the following steps only information related to the default problem type is presented to the user and a lot of preset definitions are made already. In this example session 'Dummy' is chosen as default problem type. This is done by first clicking on submenu 'Problem Types' in the main menu and selecting 'Dummy' from the selection box called 'Default Problem Type' in the submenu's page, and finally pressing 'Set'.

Next, an algorithm is created on the page accessed by link 'Algorithms' of the main menu by pressing button 'New' on this page (see figure 3.23 on page 108). On the upcoming page the algorithm for this example session is created. This algorithm will consists of only one module, namely 'Dummy', that was registered as described in subsection 3.2.2 previously. Field 'Problem Type' will already be set to 'Dummy' since this problem type was set to be the default before. The two next text input fields 'Name' and 'Description' have to be filled with the name and an optional description of the algorithm to create. In this example, the algorithm created is named `Testbed-Example`. In selection box named 'Module#1', entry 'Dummy' is chosen. Figure 3.3 shows how the values for this

example should look like.

More information about algorithms and managing problem types can be found in subsection 3.3.6 on page 107 and 3.3.3 on page 103, respectively.

3.2.5 Creating a Configuration

This example is intended to investigate the influence of the parameters `yMin` (minimum value of measurements) and `randomY` (degree of randomization of measurements) on the quality of the best solution found in terms of the quality of performance measure `best` of the dummy module. Accordingly, the algorithm just created has to be run in different fixed parameter settings differing in `yMin` and `randomY`. The submenu for defining configurations is reached via the link named 'Configurations' in the main menu (see figure 3.25 on page 111). A new configuration can be created by pressing button 'New' (see figure 3.25 on page 111). The creation of a configuration is split into three parts. First, a name and a short description are entered in the text input field named 'Name' and 'Description'. In this case the name is `Testbed-Example`. Next, the algorithm, the configuration is based on (in this case `Testbed-Example`) is selected in selection box 'Algorithm' (see figure 3.4 on page 80). After pressing button 'Set Parameters' the user can enter values for the parameters of the algorithm chosen on the next page.

Figure 3.4: Creating a configuration: Entering basic information

On the next page, the following values for the parameters stated below have to be entered (do not forget the commas, see figure 3.5 on the next page):

Configuration Testbed-Example
Algorithm Testbed-Example
Description Example from the 'Getting Started' section of the testbed manual.
Problem Type Dummy

Show/Hide Column Name Flag Type Default Condition Description

Module	**Dummy**					
Description	Dummy module for use with the testbed. Used for testing and demstration purposes.					
Name	**Flag**	**Type**	**Default**	**Values** Help	**Condition**	**Description**
function	-f, --function	INT: {1,2,3}	1		/^([+]?0*1|[+]?...	Function used to compute value...
maxMeasures	-n, --maxMeasures	INT:>0	50	30	/^[+]?[0]*[1-9]...	Maximum number of virtual meas...
maxTime	-t, --maxTime	STRING	n	110		Virtual maximum netto CPU time...
minTime	-m, --minTime	REAL:>0	0.01		/^[+]?([0]*[1-9...	Time points of measurement in ...
randomTime	-i, --randomTime	REAL:>=0	1		/^([+]?([0]*[1-...	Degree of randomization of tim...
randomType	-e, --randomType	BOOL	TRUE	On Off All Clear	/^(true|t|tt|y|...	Which probability distribution...
randomY	-r, --randomY	REAL:>=0	1	1.5,2,3,4	/^([+]?([0]*[1-...	Degree of randomization of mea...
seed	-s, --seed	INT	0		/^[+-]?[0-9]+$/...	Seed for random number generat...
tries	-x, --tries	INT:>0	10		/^[+]?[0]*[1-9]...	Number of tries (=repetitions)...
yMax	-a, --yMax	REAL:>0	10000		/^[+]?([0]*[1-9...	Maximum value for virtual meas...
yMin	-y, --yMin	REAL:>=0	0	1,1.2,1.5,2,2.5	/^([+]?([0]*[1-...	Minimum value for virtual meas...

Submit Parameter Values | Back | Cancel

Figure 3.5: Creating a configuration: Setting the parameters

Parameter	Value(s)
`Dummy_1_maxMeasures`	`30`
`Dummy_1_maxTime`	`110`
`Dummy_1_randomY`	`1.5,2,3,4`
`Dummy_1_yMin`	`1,1.2,1.5,2,2.5`

The remaining parameters are left empty; the module internal default values will work fine.

After hitting the 'Submit Parameter Values' button, on the upcoming page (see figure 3.6 on the following page) a list of all combinations of parameter values, i.e. the set of (fixed) parameter settings the configuration consists of as defined previously is shown. It is possible to save any special configuration, but at the moment there is no such need, so the whole configuration is saved with 'Create Configuration'. For more information about saving special configurations and about configuration in general see subsection 3.3.7 on page 110.

Configuration Testbed-Example
Algorithm Testbed-Example
Description Example from the 'Getting Started' section of the testbed manual.
Problem Type Dummy

Create Configuration | Back | Cancel

Fixed Parameters Settings

Dummy_1_maxTime	Dummy_1_maxMeasures	Dummy_1_randomY	Dummy_1_yMin		Name
110	30	1.5	1	Save as	Testbed-Example
110	30	1.5	1.2	Save as	Testbed-Example
110	30	1.5	1.5	Save as	Testbed-Example
110	30	1.5	2	Save as	Testbed-Example
110	30	1.5	2.5	Save as	Testbed-Example
110	30	2	1	Save as	Testbed-Example
110	30	2	1.2	Save as	Testbed-Example
110	30	2	1.5	Save as	Testbed-Example
110	30	2	2	Save as	Testbed-Example
110	30	2	2.5	Save as	Testbed-Example
110	30	3	1	Save as	Testbed-Example
110	30	3	1.2	Save as	Testbed-Example
110	30	3	1.5	Save as	Testbed-Example
110	30	3	2	Save as	Testbed-Example
110	30	3	2.5	Save as	Testbed-Example
110	30	4	1	Save as	Testbed-Example
110	30	4	1.2	Save as	Testbed-Example
110	30	4	1.5	Save as	Testbed-Example
110	30	4	2	Save as	Testbed-Example
110	30	4	2.5	Save as	Testbed-Example

Figure 3.6: Creating a configuration: List of fixed parameter settings

3.2.6 Creating an Experiment

After creating configurations, an experiment can be specified in the 'Experiments' submenu of the main menu (see figure 3.26 on page 116). A new experiment is created by pressing button 'New'. First, the name and a description of the experiment is entered in text input fields 'Name' (in case of the example session again `Testbed-Example`) and 'Description'. Next, the configurations and problem instances that are to be used in the experiment are selected in the selection lists named 'Configurations' and 'Problem Instances', respectively (see figure 3.7 on the next page). The check box for the online process information is left untouched in this example and the previously imported problem instance `100.Dummy.dat` is selected by clicking on it. In the selection list for configurations, configuration `Testbed-Example` is chosen. Selected entries of selection lists will be highlighted. By pressing button 'Create Experiment' the experiment is stored in the database. Note that no job has been created or started yet.

Note that if eligible, more than one entry can be selected in a selection list by holding down key 'Control/Ctrl' on the keyboard while clicking the entries that are to be selected.

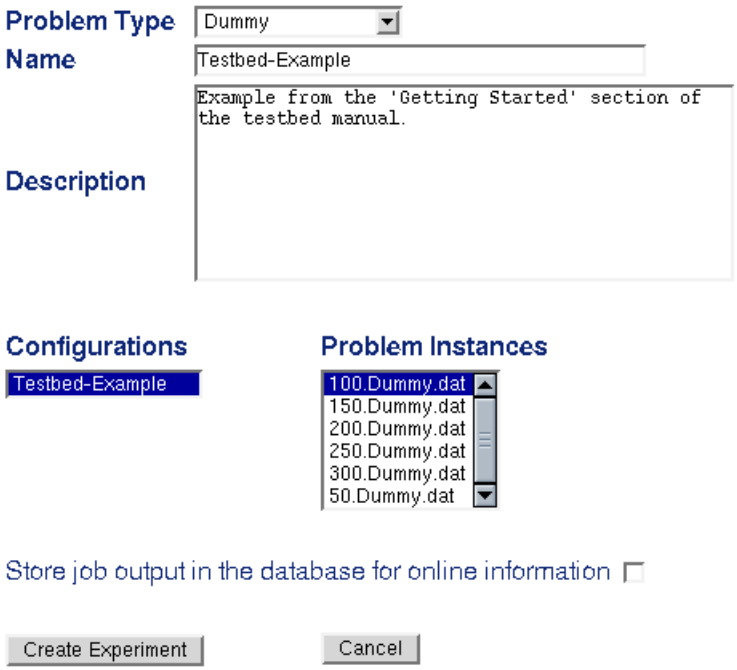


Figure 3.7: Creating an experiment

If a number of entries located successionally in a selection list are to be selected, the first entry can be clicked, next key 'Shift' is hold while the last entry of entries to be selected is clicked. This will select all successional entries bordered by the first and last entry selected.

The jobs resulting from the experiment can be created and started on the next page where the user is automatically lead to (see figure 3.8 on page 86). Before starting jobs (in this case 20), the testbed presents a list of all jobs of an experiment, i.e. a list of all combinations of (fixed) parameter settings from the configuration(s) of the experiment with the problem instances of the experiment. The user can get an overview how many job will be run and how long it may take. If the user is sure to start the jobs, a priority can be entered and the hardware the jobs should run on is chosen at the end of the job list (if no input is made, a default priority of 50 is taken, see subsection 3.3.8 on page 116 for more details). By pressing button 'Start Experiment' the 20 jobs of this example session are really generated by storing their specification into the testbed database and by putting them into the execution queue.

After the jobs have been generated by the testbed, a list of all jobs of the just started experiment is displayed. See figure 3.9 on page 86 for the job list.

The specification of experiments is discussed further in subsection 3.3.8 on page 116 while the management of jobs is treated in subsection 3.3.9 on page 121.

3.2.7 Running an Experiment

A bunch of job servers, which can be started via the CLI on the client machines in a network of computers, execute the jobs from the job execution queue. These job servers retrieve the problem instances from the database, execute the modules of each algorithm in the defined order while managing the inter module data transfer via temporary files and store the result of the last module back to the database. Starting a server is done by calling on the CLI on the according machine

```
testbed server
```

The job server will directly start the first waiting job that can be found in the execution queue. An ordering of job execution is not possible (compare to section 3.3.14 on page 134 and subsection 3.3.9 on page 121). While a job is running dots will be printed to the screen to show that the module's process is still running. The output will look like the following:

```
Example-Module>testbed server
/============================================================\
====================== Working on Job #780 ======================
\============================================================/

   Module: Dummy
------------------
Executing: '~/Testbed/bin/i386/linux/Dummy/Dummy
            --finallyWait 0 --finallyFail 0
            --maxMeasures "30" --maxTime "110" --randomY "1.5" --yMin "1"
            --input "/tmp/testbed/jobs/780/100.Dummy.dat"
            --output "/tmp/testbed/jobs/780/output.dat"'
Progress: .

Execution of module succeeded!

/============================================================\
====================== Job #780 succeeded! ======================
\============================================================/
                              .
                              .
                              .
/============================================================\
====================== Working on Job #799 ======================
\============================================================/

   Module: Dummy
------------------
```

```
Executing: '~/Testbed/bin/i386/linux/Dummy/Dummy
            --finallyWait 0 --finallyFail 0
            --maxMeasures "30" --maxTime "110" --randomY "4" --yMin "2.5"
            --input "/tmp/testbed/jobs/799/100.Dummy.dat"
            --output "/tmp/testbed/jobs/799/output.dat"'
Progress: .

Execution of module succeeded!

/============================================================\
====================== Job #799 succeeded! ======================
\============================================================/

No jobs in queue, waiting ...
No jobs in queue, waiting ...
```

In case of this example session, it will take about no time to complete all 20 jobs. When all jobs are finished, i.e. no more waiting jobs can be found in the database, a job server will report that there are no more jobs to execute. A job server can be aborted with 'Control-C' on the command line. Note that the order in which the jobs are executed need not be the same as implied by their numbers.

As soon as all jobs from the example experiments have been run, clicking on the 'Reload' button of the page from figure 3.9 on the next page will yield the same page, however this time with more information about the job execution (see figure 3.10 on page 87).

3.2.8 Evaluating an Experiment

After all jobs of the example experiment have been completed, the evaluation of the example experiment can be started. The output of the jobs run during the example session experiment can be extracted, viewed and analyzed in submenus 'Data Extraction' and 'Data Analysis' (see figures 3.11 on page 88 and 3.31 on page 130). The general procedure is as follows: First, a script for extracting data from the result of jobs is needed. This script will extract data from the job results. In doing so, the script is applied to each job result of an experiment once. The combined data extracted then can be viewed in the testbed, exported to the file system, or internally transfered for subsequent analysis by the R package. In the latter case, an R or rather analysis script that is to be applied to the output of the extraction effort has to be selected, too. For this example, four kinds of evaluations will be conducted:

1. Computation of summarizing statistics over the results of the single tries for each job.

Experiment 'Testbed-Example' added

Experiment Testbed-Example
Problem Type Dummy
Status Waiting
Description Example from the 'Getting Started' section of the testbed manual.

Configurations **Problem Instances**
Testbed-Example 100.Dummy.dat

20 jobs will be started

Parameters							
JobNo	Dummy_1_maxTime	Dummy_1_maxMeasures	Dummy_1_randomY	Dummy_1_yMin	Problem Instance	Status	Action
	110	30	1.5	1	100.Dummy.dat	Waiting for Creation	
	110	30	1.5	1.2	100.Dummy.dat	Waiting for Creation	
	110	30	1.5	1.5	100.Dummy.dat	Waiting for Creation	
	110	30	1.5	2	100.Dummy.dat	Waiting for Creation	
	110	30	1.5	2.5	100.Dummy.dat	Waiting for Creation	
	110	30	2	1	100.Dummy.dat	Waiting for Creation	
	110	30	2	1.2	100.Dummy.dat	Waiting for Creation	
	110	30	2	1.5	100.Dummy.dat	Waiting for Creation	
	110	30	2	2	100.Dummy.dat	Waiting for Creation	
	110	30	2	2.5	100.Dummy.dat	Waiting for Creation	
	110	30	3	1	100.Dummy.dat	Waiting for Creation	
	110	30	3	1.2	100.Dummy.dat	Waiting for Creation	
	110	30	3	1.5	100.Dummy.dat	Waiting for Creation	
	110	30	3	2	100.Dummy.dat	Waiting for Creation	
	110	30	3	2.5	100.Dummy.dat	Waiting for Creation	
	110	30	4	1	100.Dummy.dat	Waiting for Creation	
	110	30	4	1.2	100.Dummy.dat	Waiting for Creation	
	110	30	4	1.5	100.Dummy.dat	Waiting for Creation	
	110	30	4	2	100.Dummy.dat	Waiting for Creation	
	110	30	4	2.5	100.Dummy.dat	Waiting for Creation	

Start Experiment Priority: On same Hardware

Figure 3.8: Starting an experiment

Jobs of Experiment 'Testbed-Example' added.

Overview Jobs - Showing 1 - 10 of 20

Category Show All Testbed-Example

1 - 10,11 - 20

Timestamps

JobNo	Experiment	Configuration	Parameters	Restarts	Status	Action
978	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
979	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
980	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
981	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
982	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
983	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
984	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
985	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
986	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	
987	Testbed-Example	Testbed-Example	Dummy_1_maxMeasures ...	0	Waiting	

Dump all 20 Jobs

Done

Figure 3.9: List of jobs

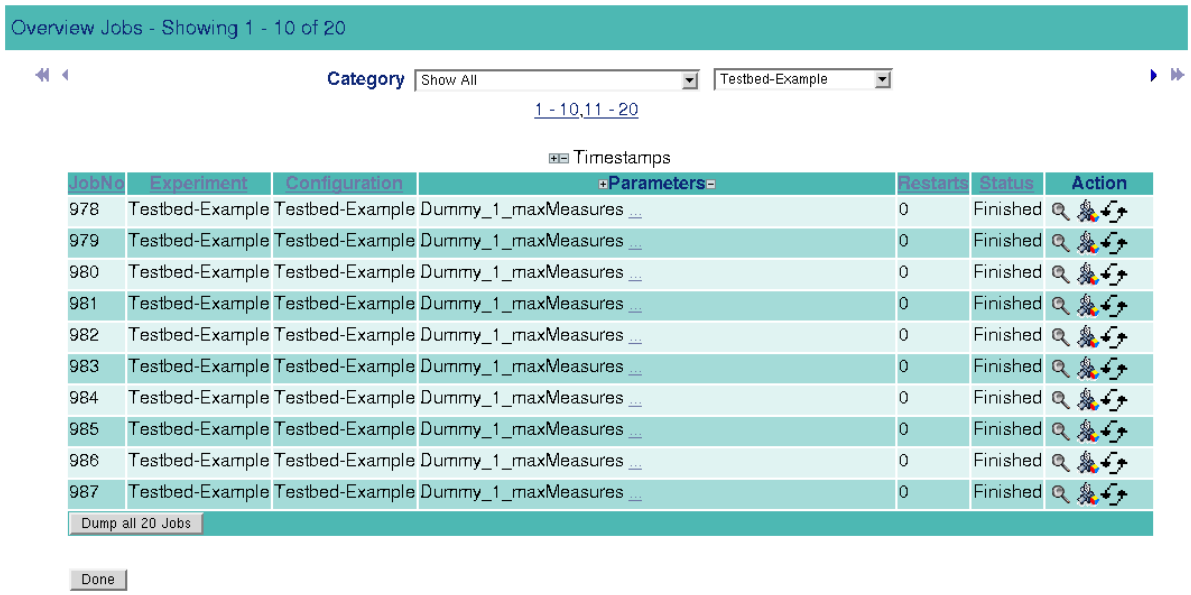

Overview Jobs - Showing 1 - 10 of 20
Category
Show All
Testbed-Example
1 - 10,11 - 20
Timestamps
JobNo
Experiment
Configuration
Parameters
Restarts
Status
Action
978 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
979 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
980 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
981 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
982 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
983 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
984 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
985 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
986 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
987 Testbed-Example Testbed-Example Dummy_1_maxMeasures ... 0 Finished
Dump all 20 Jobs
Done

Figure 3.10: List of finished jobs

2. Statistical testing with respect to significant differences in the mean and median performances of the various settings for parameter `yMin` with respect to performance measure `best`.

3. Plotting of box plots, one for each level of randomization as configured through parameter `randomY`. Each box plot will compare the results for different values of parameter `yMin` for a specific level of parameter `randomY`.

4. Plotting of trade-off curves of solution quality in terms of performance measure `best` vs. runtime, i.e. a plot of development of performance measure `best` over time.

For this example session, the standard data extraction scripts can be used. The data extraction scripts for the example session can be imported in the 'Scripts' submenu of submenu 'Data Extraction', by first clicking on button 'Browse' (see figure 3.11 on the following page). This will open the file browser of the web browser. The example scripts are located in directory `DOC_DIR`*/example_session/extraction_scripts* and are named *Summary-Last-Of-Each-Try.X.xml*, *Extract-Last-Of-Each-Try.X.xml*, and *Averaged-Trade-off-Curve.X.xml*. After selecting a file with the help of the file browser, the script can be imported by pressing button 'Import Script' (see subsection 3.4.3 on page 139 for detailed information about how to import XML files).

Extraction script `Extract-Last-Of-Each-Try` will extract all the best, i.e. minimal values for performance measure `best` of all tries of a job. This will yield a list of 10 best solution values for each job. These 10 solution values per job can then be used as input for a statistical test or a box plot. Extraction script `Averaged-Trade-off-Curve` will

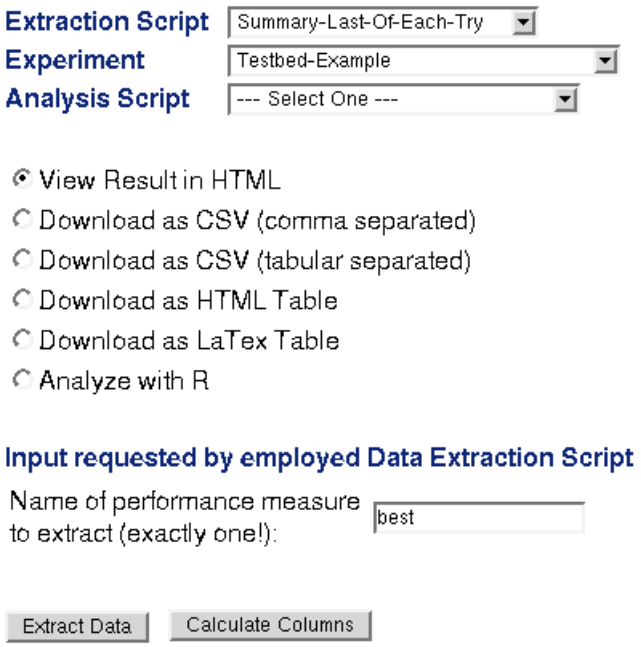


Figure 3.11: Extracting data from job results

extract the runtime development of performance measure `best`, averaged over the tries for each job. Both scripts output can be viewed, but the output actually is intended for input to analysis scripts doing plots or statistical testing. Finally, extraction script `Summary-Last-Of-Each-Try` need no further processing, since it will compute summarizing statistics such as 'Mean', Median', Variance', and so on for each job computed over the tries of each job.

A broad description of the data extraction language the extraction functioning is given in section 4.3 on page 192 while general information about the management of extraction scripts is given in subsection 3.3.10 on page 125.

The analysis script employed in this example are parameterized version of some generic scripts. They have to be imported the same way as the extraction scripts, this time in submenu 'Script' of submenu 'Data Analysis' (see figure 3.31 on page 130). The files used in this example session are located in `DOC_DIR`*/example_session/analysis_scripts/* and are named *Testbed-Example–Stat-Tests.R.xml*, *Testbed-Example-Boxplots.R.xml*, and *Testbed-Example-Plot-Curves.R.xml*.

For all evaluations, experiment `Testbed-Example` is kept selected in selection box 'Experiment'. For the first evaluation, script `Summary-Last-Of-Each-Try` previously imported is selected in the selection box named 'Extraction Script'. A user input request will show up (see figure 3.11), which, however, needs no changes, since the default value `best` works fine for this example. The result of the extraction effort can be viewed by checking check box 'View Result in HTML' and subsequently pressing button 'Extract

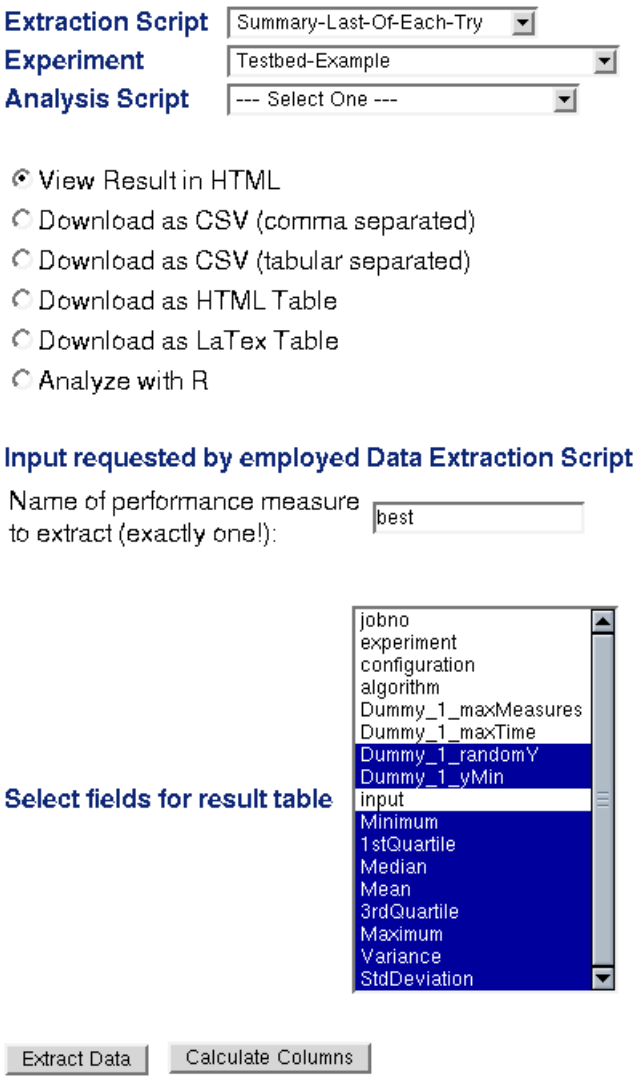


Figure 3.12: Data extraction: Calculating columns

Data'. After some processing time, the output of the data extraction effort is presented as an table on a new page containing a summary of the statistics `best` over the tries of each job. Each row represents one job, each column represents one attribute of the job, the columns ranging from the job's algorithm over its parameter settings to the statistics. Since not all columns are always of interest, some columns can be discarded. This is done by pressing button 'Calculate Columns' before starting the extraction process with button 'Extract Data'. A selection list with all columns will be displayed (see figure 3.12 on the preceding page). Now, the relevant columns `Dummy_1_randomY`, `Dummy_1_yMin`, `Minimum`, ..., `StdDeviation` can be selected by clicking on them while holding down key 'Control/Ctrl'. Pressing button 'Extract Data' now will yield a smaller result table as depicted in figure (see figure 3.13).

Dummy_1_randomY	Dummy_1_yMin	Minimum	1stQuartile	Median	Mean	3rdQuartile	Maximum	Variance	StdDeviation
1.5	1	1.00100	1.08354	1.60441	1.552992	1.89428	2.00100	0.162486470484	0.403096105767
1.5	1.2	1.20100	1.28354	1.80441	1.752992	2.09428	2.20100	0.162486470484	0.403096105767
1.5	1.5	1.50100	1.58354	2.10441	2.052992	2.39428	2.50100	0.162486470484	0.403096105767
1.5	2	2.00100	2.08354	2.60441	2.552992	2.89428	3.00100	0.162486470484	0.403096105767
1.5	2.5	2.50100	2.58354	3.10441	3.052992	3.39428	3.50100	0.162486470484	0.403096105767
2	1	1.17507	1.56303	1.74671	1.692774	1.97461	2.00100	0.0867788004711	0.294582417111
2	1.2	1.37507	1.76303	1.94671	1.892774	2.17461	2.20100	0.0867788004711	0.294582417111
2	1.5	1.67507	2.06303	2.24671	2.192774	2.47461	2.50100	0.0867788004711	0.294582417111
2	2	2.17507	2.56303	2.74671	2.692774	2.97461	3.00100	0.0867788004711	0.294582417111
2	2.5	2.67507	3.06303	3.24671	3.192774	3.47461	3.50100	0.0867788004711	0.294582417111
3	1	1.37052	1.93399	2.01426	2.128794	2.52284	2.84144	0.192457577293	0.438699871545
3	1.2	1.30502	1.85722	2.18327	2.239969	2.51005	3.31064	0.379939428766	0.61639226858
3	1.5	1.60502	2.15722	2.48327	2.539969	2.81005	3.61064	0.379939428766	0.61639226858
3	2	2.10502	2.65722	2.98327	3.039969	3.31005	4.11064	0.379939428766	0.61639226858
3	2.5	2.60502	3.15722	3.48327	3.539969	3.81005	4.61064	0.379939428766	0.61639226858
4	1	1.47203	1.99643	2.33052	2.599075	3.07531	4.14247	0.682122709494	0.825907203925
4	1.2	1.67203	2.19643	2.53052	2.799075	3.27531	4.34247	0.682122709494	0.825907203925
4	1.5	1.97203	2.49643	2.83052	3.099075	3.57531	4.64247	0.682122709494	0.825907203925
4	2	2.47203	2.99643	3.33052	3.599075	4.07531	5.14247	0.682122709494	0.825907203925
4	2.5	2.97203	3.49643	3.83052	4.099075	4.57531	5.64247	0.682122709494	0.825907203925

Figure 3.13: Data extraction: Viewing results

Using the 'Back' button of the web browser or the entry 'Data Extraction' in the main menu, the 'Data Extraction' submenu can be reached again. In order to directly process the data extracted with R, an analysis script can be selected in selection box 'Analysis Script'. By checking check button 'Analyze with R' the data extracted is conveyed directly to the R package upon hitting button 'Extract Data' and the chosen analysis script is run on this data. Before processing the data, it can, of course, be viewed as well.

For the second evaluation, instead of choosing 'View as HTML' and extraction script `Summary-Last-Of-Each-Try`, extraction script `Extract-Last-Of-Each-Try` for extracting the best results per try is chosen as well as analysis script `Testbed-Example--Stat-Tests` for doing statistical testing. Check box 'Analyze with R' is checked. The user input, again, is set to the suitable default `best`. The results of the analysis script processing

of the data will be displayed on a new page. The analysis script employs two statistical tests under the null hypothesis that the solution quality is the same not matter which level of parameter `yMin` was chosen. The results will look as follows:

```
=============
Test group 1:
=============
 Dummy.1.randomY: 1.5
 Samples:
    1    Dummy.1.yMin: 1
    ===> 1
    2    Dummy.1.yMin: 1.2
    ===> 1.2
    3    Dummy.1.yMin: 1.5
    ===> 1.5
    4    Dummy.1.yMin: 2
    ===> 2
    5    Dummy.1.yMin: 2.5
    ===> 2.5

 Parametric test:
 ================
 Method:          Analysis of variance
 Statistic:       F  (4,45)
 F-value:         44.16677
 Pr(>F):          4.996004e-15
 df:     Samples:        4
         Residuals:      45
 Hypothesis 'means are all equal' REJECTED on basis of given critical
 p-value of 0.01!

 Non-parametric test:
 ====================
 Method:          Kruskal-Wallis rank sum test
 Statistic:       Kruskal-Wallis chi-squared
 Value:           39.03309
 df:              4
 p-value:         6.857664e-08
 Hypothesis 'medians are all equal' REJECTED on basis of given critical
 p-value of 0.01!

 Do pairwise testing
 ===================
    Testing 1 (1) vs. 1.2 (2):
```

```
-------------------------
   Parametric test:
   -----------------
      Method:          Welch Two Sample t-test
      Statistic:       t
      t:               -3.322204
      df:              16.55415
      p-value:         0.004151805
      Hypothesis 'true difference in means is equal to 0' REJECTED on
      basis of given critical p-value of 0.01!

   Non-parametric test:
   --------------------
      Statistic:       Kruskal-Wallis chi-squared
      Value:           6.655758
      df:              1
      p-value:         0.009883594
      Hypothesis 'medians are equal' REJECTED on basis of given critical
      p-value of 0.01!
Testing 1 (1) vs. 1.5 (3):
-------------------------
                .
                .
                .
```

For the third and fourth evaluation, data has to be extracted and saved to disk, before it can be analyzed, e.g. before it can be plotted. Any plotting within the testbed has to take the detour of saving the plot data temporarily to the file system. The data for the third evaluation is created by selecting extraction script `Extract-Last-Of-Each-Try` with performance measure `best` in the text input field and check box 'Download as CSV (comma separated)' checked. Using button 'Calculate Columns', columns `Dummy_1_randomY`, `Dummy_1_yMin`, `try`, and `best` are selected. After pressing button 'Extract Data', the browser asks where to store the CSV file. The file can be stored to some temporary directory under name *Testbed-Example-Box plot.csv*. The data for the fourth evaluation has to be stored the same way, this time using extraction script `Averaged-Trade-off-Curve`, file name *Testbed-Example-Curve.csv*, and columns `Dummy_1_randomY`, `Dummy_1_yMin`, `Time`, `Minimum`, `Mean`, `Maximum`, `Std.deviationLower`, and `Std.deviationUpper`. Directory `DOC_DIR`*/example_session/download/* contains example downloads with the files names just mentioned.

After having saved the data extracted, the analysis scripts for plotting can be selected in submenu 'Data Analysis' (see figure 3.14 on the facing page). Selection box named 'Analysis Script' can be used to select the analysis script to apply, in this case in succession `Testbed-Example-Boxplot` and `Testbed-Example-Plot-Curve`. Using button

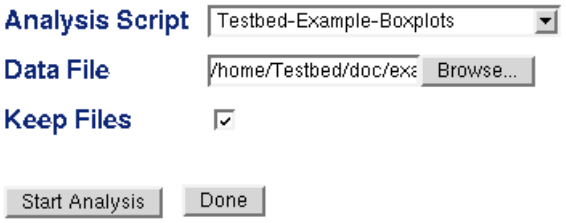


Figure 3.14: Analyzing data

'Browse' right next the text input field named 'Datafile', the two files just stored can be selected, first file *Testbed-Example-Box plot.csv*, the file *Testbed-Example-Curve.csv*. Checking check box 'Keep Files' will tell the testbed to keep any files that are created by the scripts and make them accessible for the user. In this example this is necessary, since the scripts will create graphic files containing the plots. Clicking button 'Start Analysis' start the script on the selected data. A new page will come up with the text output of the analysis script just run, e.g. some error or status message or warnings (see figure 3.15).

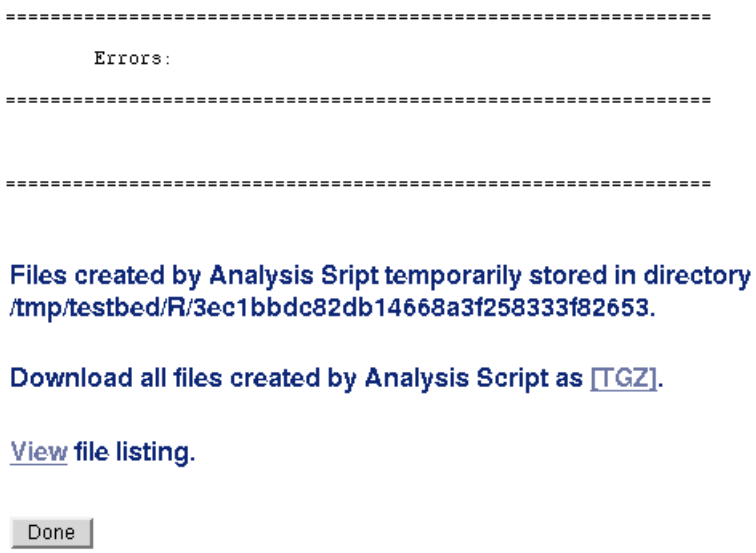

Errors:
Files created by Analysis Sript temporarily stored in directory
/tmp/testbed/R/3ec1bbdc82db14668a3f258333f82653.
Download all files created by Analysis Script as [TGZ].
View file listing.
Done

Figure 3.15: Analyzing data: View results

At the bottom of this page, clicking link 'View (file listing)' will lead to yet another page which lists the single plots (see figure 3.16 on the following page). Any of the links ending with '.png' contains a box plot, one for each level of parameter `randomY`. each such plot can be viewed within the web browser. When using the second analysis script to produce trade-off curves, the links will lead to graphics of trade-off curves, one for each combination of each level of parameters `randomY` and `yMin`.

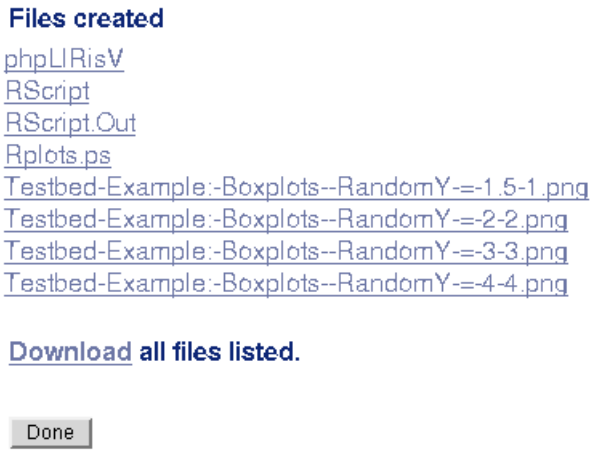


Figure 3.16: Analyzing data: File listing

3.3 Testbed in Detail

This section presents a detailed description of the testbed and its functionality. The testbed is structured according to the different components of experimentation as described in section 2.3 on page 13. This structure resembles the different stages of the process of experimentation. It also reflects the conceptual and data objects of different types[9] and purposes that are involved by the process and are thus contained in the testbed such as problem types, problem instances, algorithms, configurations, experiments, jobs, and data and analysis scripts.

The main menu of the testbed (see figure 3.1 on page 78), always located on the left side of the testbed web pages, also reflects this structure. Each submenu represents one step and aspect of experimentation in terms of grouping similar data together in a submenu. For example, the submenu representing algorithms, 'Algorithms', displays the algorithms or rather algorithm specifications contained in the testbed to the user and provides means to manipulate these. Submenus manage groups of similar data. The single (data) objects a submenu presents are called *entries*[10]. That way, they provide a coarse overview over the objects contained in the testbed they represent and provide some general functionality such as editing, deleting, and creating new objects. Beside this common functionality, each submenu has special operations that can be performed on the entries.

In this section, the common functionality of submenus is described first together with common operations that can be realized with the entries of each submenu. Some general information explaining details about object dependencies will be given also. In the second part of this first subsection, the handling and navigation within submenus is explained. Next, a detailed description of each submenu is given in the following subsections with one subsection per submenu. The section concludes with a discussion of how to check the testbed's status and of how to organize and handle different types of hardware within the testbed when run in a network. Finally, the the command line interface of the testbed that provides important functionality that can not be implemented via a web front end is presented and treated in detail.

3.3.1 User Input

Before plunging into the details, some basic information with respect to how the user can enter information in web pages is given. The user can enter information on web pages in different ways. Any user input is requested via various types of input fields.

[9] The notions of data type, object type, kind of object, type of object, type of data are used interchangeably throughout this text. Compare with begin of section 2.4 on page 40 and subsection 2.1.2 on page 7

[10] The notions entry and object are used interchangeably through this document, essentially meaning the same

These are shortly discussed next:

Text Input Field: The user can enter arbitrary text, if such an input field is active, e.g. if being clicked by the user. Text can be highlighted with the mouse pointer or by holding down key `SHIFT` while moving the cursor. Highlighted text can be copied to the clipboard with keys 'Control-C', it can be cut to the clipboard with keys 'Control-X' and finally, text copied to the clipboard can be inserted at the current cursor position with keys 'Control-V'. Undo of operations can be triggered with keys 'Control-Z'.

Selection Box: The user can expand input fields of this type by clicking on the downward arrow on the right border of these input fields. This will drop down a list of choices. By clicking on one of the choices the user selects it. The dropped down list vanishes again, but can be expanded again, too.

Selection List: A selection list presents a list of choices to the user, too. However, the list will be already expanded, possibly providing a scroll bar, if the place on a web page designated to the list is not sufficient to display all choices at once. The user can select several choices at once by highlighting them. Elements of the list are highlighted by clicking on them. By holding down key 'Control/Ctrl', newly clicked elements will be highlighted, *too*. Holding down `SHIFT` while clicking on an element highlights all elements in between the newly clicked element and the next element above the newly clicked element in the list that is highlighted, too.

Radio Button: Radio buttons are used to enforce the selection of exactly one or none of a couple of choices. They are little circles, and if selected by clicking on them, they are filled with a solid bullet. Exactly one of the choices presented to the user in a coherent list of radio buttons can be selected.

Check Box: Check boxes are displayed as little squares and work almost as radio buttons. They can be clicked in which case a small check mark will appear in the check box. However, they are completely independent from any other check boxes or input fields.

Button: A button is used by the user to trigger an action. Buttons come in the form of grey shaded rectangles that are labeled with an indication which action they are supposed to start.

3.3.2 Submenus

This section explains in the first part labeled 'Submenu Functionality' some common notions and functionality of all submenus. In the next part, 'Submenu Handling', the means for the user to interact within submenus are covered.

Submenu Functionality

Each type of object in the testbed resembles one conceptual component or rather aspect of the process of experimentation as pointed out in subsection 2.1.2 on page 7 and in section 2.4 on page 40. Each submenu now represents a special type of object contained in the testbed. The represented type will also be called the *submenu's type*. All objects or rather entries of this type will be displayed by a submenu in the form of a table. Each entry occupies one row. The columns provide the atomic pieces of information for the entries.

Usually, the number of objects of a submenu's type in total is too large to fit on one page. With the help of so called *filters*, the subset of entries that actually are of interest and which are to be displayed can be confined. Filters pose restriction on each entry and only entries meeting these restrictions are not filtered out and will actually be available for display. Filters can be the so called *current search filter* (see subsection 3.5.1 on page 146), a *category* or *category filter* (see subsection 3.5 on page 146), an *experiment filter*, a *problem type filter* or *regular expression filter*. Experiment filters simply filter out all entries that do not belong or are not related to the chosen experiment, while regular expression filters are constructed by supplying a regular expression which is used to filter out all entries whose name does not match the entered regular expression. The regular expression that can be entered here are the same and work the same way as the ones used within the search filter generation tool from subsection 3.5.1 on page 146. Their syntax and their functioning is described in detail in paragraph 'Wildcards and Regular Expressions' on page 162 in subsection 3.5.1. Problem type filters work the same way as experiment filters only filtering according to the problem type of the entries. Category filters are essentially queries to the testbed database that have been stored. Typically, either experiment or problem type filters are available and sometime some filters are missing in certain submenus for inapplicability reasons. The submenus representing problem instances, modules, algorithms, configurations, and experiments feature problem type filters. The submenu for jobs features a experiment filters, while submenus for scripts do not feature either filter type: The filters selectable here do not have any effect yet. Regular expression filters applied to the name of entries are featured by any submenu displaying entries. Further information is given in the subsection devoted to the particular submenus.

Usually, even the subsets of entries obtained through the application of filters in a submenu will not fit on one page. Therefore, all entries remaining after application of a filter, also called *the entries available for displays*, are distributed equally over multiple pages in an order specified by the user. These pages containing subsets of all entries available for display are called *segments*. The user can, by clicking on the according column name, indicate to sort the entries with respect to the order that is induced by the column selected. In case of textual contents of a column, the lexicographical order is used, columns containing numbers are sorted according to the common order of

numbers. Not all columns can be used for ordering. The columns suitable are indicated as hyperlinks, i.e. they are underlined in most browsers (see figure 3.17 on page 100, item **6.1**). The maximum number of entries per page that are actually displayed can be changed in the submenu labeled 'Preferences' (compare to subsection 3.3.12 on page 132 under keyword 'Max Matches'). Navigation through the segments is explained in the next subsection.

In many submenus, entries can be added, edited, copied and edited, deleted, and exported. The applicability of these operations differ from submenu to submenu, i.e. from object type to object type. However, some general rules apply. When new entries are added, or existing entries are edited and copied, or edited, a form on a new page is provided by the testbed with buttons, check boxes, and text inputs fields to fill in the information needed. After entering the information, the user submits the data to the testbed by pressing a special button. The testbed validates the user input for consistency and errors, e.g. it checks whether names are unique, whether they do contain forbidden characters, and so on. If data is missing or if erroneous or inconsistent data was entered by the user, the testbed will output a message and display the form again with the old input already filled in. The user now can change the data or add missing data and submit the completed form again. Any operation can be aborted by pressing button 'Cancel' (see figures 3.18 on page 103, 3.19 on page 105, or 3.24 on page 108 for an example).

Entries or rather objects typically are identified uniquely within the testbed or rather within the testbed's database by their names. Objects in the testbed can depend on each other. For example, an experiment specification depend on one or more configurations (identified by their names), which in turn depend on some algorithm specifications. Configurations in turn not only do depend on being able to uniquely identify their underlying algorithm, but also do depend on specific aspects of the algorithm they configure such as visible or configurable parameters. It follows that deleting or renaming objects, or changing the specification of objects can corrupt dependencies. In order to avoid such a corruption, objects can not be renamed. Specification details of objects potentially having dependencies attached to these details such as algorithm parameters, can not be changed either, while deletion of an object automatically will delete all dependent objects from the testbed, too, possibly transitively. This is, because otherwise the database might not be in a consistent state after deletion. Objects can, however, be copied and edited. This operation will open a form to fill in the information needed for specifying an object of a given type with the information of the original already filling the input fields and boxes. This information then can be changed, in particular the name *must* be changed, of course.

Throughout the submenus, the same symbols, also called *icons* have been used to indicate common operations. The actions applicable to an entry are indicated by the presence of the representing icon in the last column of each submenu called 'Action' as can be seen

Icon	Action	Explanation
	Edit	On the upcoming page the user can edit the entry. Not all entries can be edited (such as jobs or algorithms) and not all aspects of an entry's specification can be edited such as the name.
	Edit Description	If only the description of an entry can be changed, this icon will appear instead of the 'Edit' icon.
	Copy & Edit	In order to change entries that might have dependencies attached or in order to copy an entry together with its specification, this icon can be clicked. On the upcoming page, the form for creating entries of the submenu's type will be displayed with the input fields and boxes already filled with the settings from the original. Only the name *must* be changed.
	Delete	Using this icon will delete an entry. The testbed will display the details of the entry to be deleted on a new page and will ask for confirmation before the entry is permanently removed from the testbed. Pressing button 'Delete' will really delete the entry, pressing button 'Cancel' will lead back to the submenu. Note that dependent objects will be removed automatically, too!
	Show details	Clicking this icon will show a detailed description of the entry on an extra page. This page can not be edited even if it sometimes look like the forms used to add a new entry to the testbed.
	Export as XML file	If the web browser can read XML, a new page will open after clicking on this icon. The file can then be saved through the browser's 'Save File' function. In this case the 'Back' button of the browser must be used to get back to the testbed. If the browser can not read XML, the file is saved directly to the disk by the browser. In both cases, the browser will open a file browser sub window for storage of the export file.
	Set category as current search filter	Sets the category chosen as the current search filter. Only applicable for categories in the 'Categories' submenus and in submenu 'Global Categories' (see subsection 3.5.2 on page 165).

Table 3.2: Common icons and actions

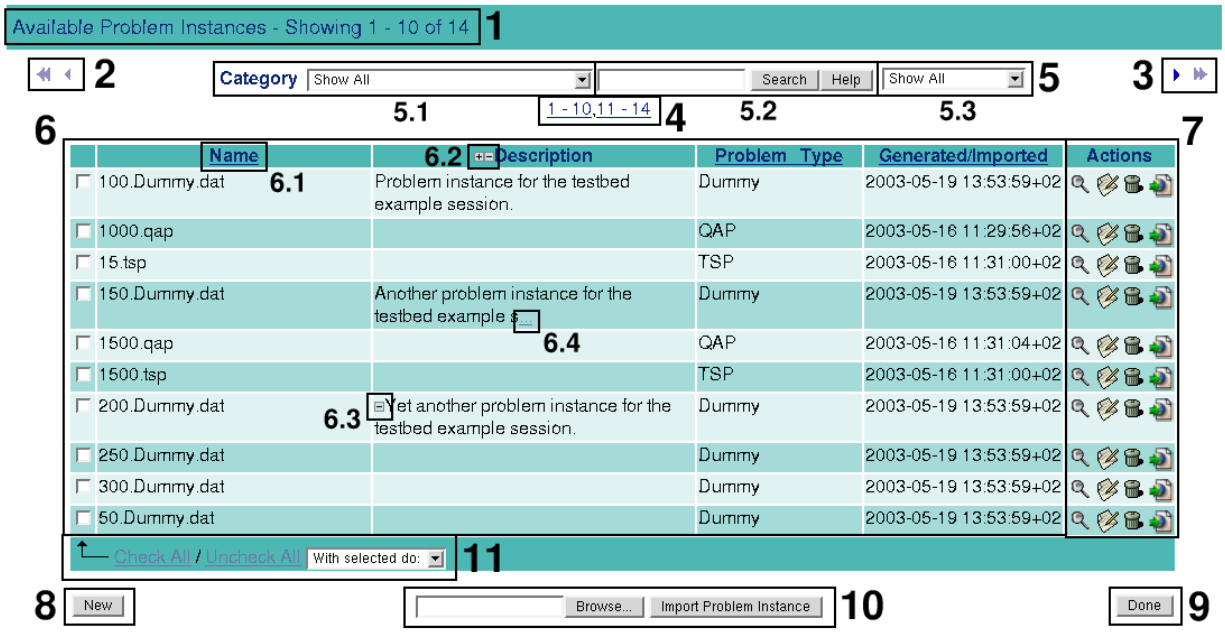


Figure 3.17: Submenu appearance

in figure 3.17, item **7**.

Table 3.2 on the page before list all common icons of column 'Action' are presented and explained. The jobs submenu allow for special operations to be performed on their entries, since these are not created directly by the user. These operations, together with their graphical representation, will be discussed in subsection 3.3.9 on page 121.

One submenu is devoted to the user manual. It is called 'User Manual' and will lead to a page which contains this document in the form of an HTML page. The submenu 'PDF' of the 'User Manual' submenu contains this document in the form of a PDF document (see figure 3.1 on page 78).

Submenu Handling

All information is presented to the user via web pages by the testbed. The interaction with the testbed is similar to browsing the Internet. Almost all submenus of the testbed feature a similar layout and usually all submenus have the same elements. An example submenu can be viewed in figure 3.17. The individual elements of submenus as indicated by the integer labels are described next.

1 On top of the page, the number of entries currently displayed by the current page and the number of entries available for display are shown. Note that the number of entries available for display need not always be equal to the total number of objects of the submenu's type existing in the testbed: With the help of filters the number

of entries to be displayed can be decreased (compare with the information given in subsection 3.5.1 on page 146 and the previous paragraph for more information). Only entries matching the criteria set by the filters (**5**) are available for display.

2 Pressing the fast rewind icon will display the first segment of entries available for display. Pressing the single rewind sign will lead to previous segment.

3 The same functionality as in 2, works the other ways round.

4 In order to directly switch to certain segments without having to scroll sequentially through the segments, the user can click on the desired segment to get there immediately.

5 Filters determine a subset of all entries of a submenu that is to be displayed instead of making all entries available for display. These filters can either be a predefined filter in the form of an experiment or problem type filter (selectable in item **5.3**), the current search filter, a category (both selectable in item **5.1**) or can be constructed as a regular expression (selectable in item **5.2**). More information about categories can be found in subsection 3.5 on page 146, search filters and the current search filter are discussed in subsection 3.5.1 on page 146, while experiment, problem type and regular expression filters are introduced in the previous paragraph.

Filters can be combined. They will be logical AND connected, i.e. they are applied simultaneously and only entries *not* filtered out by *any* filter applied will be available for display. If no specific filter is supposed to apply, entry 'Show All' has to be selected as category and experiment or problem type filter, while the text input field for entering regular expressions has to be cleared. Recall from the preceding section that sometimes filters are not applicable in certain submenus. Which submenu supports which type of filters is discussed in the respective submenu. Button 'Help' can be used to get online help for the construction of regular expressions. Button 'Search' starts the regular expression filter process. All other filter processes are started as soon as a filter is selected in one of the filter selection boxes.

6 All information that is necessary in order to identify an entry is shown in one line for each entry. It is also possible to change the order in which the entries are displayed by clicking on the column names. The column that is selected serves as the ordering criteria. Note that not all columns can serve as an ordering criteria. Those that are eligible are indicated by underlined names as in item **6.1**. Successive clicks on an underlined column name switches the ordering between ascending and descending.

The information displayed by some columns such as column 'Description' often does not fit into one single line or would stretch the column too wide. For this

reason, some columns are also equipped with a pair of small buttons in the form of a + sign (⊞) and a − sign (⊟), see item **6.2**. The + sign is used to expand a column: The whole colum contents for each entry is shown, possibly stretching the whole table substantially. The − sign is used to implode this information again in order decrease the circumference of the table. Single entries can be expanded and imploded independently of the other entries, too, by pressing the underlined dots (...) at the end of a column in imploded state (item **6.3**) and by pressing the preceding − sign in expanded state (item **6.4**).

7 The last column of each row indicates all possible actions that can be performed on an entry by displaying a little icon per action possible. For example, it is possible to export an entry to XML, edit an entry, or delete an entry. An overview of the most common actions is given in table 3.2 on page 99. Which actions are available in each submenu is described in the subsections covering the individual submenus.

8 The 'New' button can be used to create a new entry or rather object of the submenu's type.

9 Button 'Done' leads the user to the lastly visited page.

10 Beside export to XML, each submenu features an import facility. After selecting an XML file to import entries with the 'Browse' button from the local file system (the browser will open a file browser window and after the selection of a file, the file name is shown in the field left to the 'Browse' button), the user can actually import this file into the testbed by pressing button 'Import ...'. A new page will appear with messages describing the outcome of the import effort. Using button 'Done' on this page leads back to the last page. Import of exported XML files comes with some subtleties. It is strongly recommended to read the details in subsection 3.4.3 on page 139.

11 Typically, all actions apply to one single entry only. Some actions, however, frequently have to be performed on a number of entries. Clearly, it would be quite cumbersome, if the user had to invoke an action for each entry individually. For this reason, actions delete and export to XML can be performed on multiple entries at once, if applicable in a submenu. The left most column of some submenus providing delete and/or export functionality contains a check box for each entry. Checking the check box indicates that the entry is subject to the action that can be selected in the selection box at the bottom of the submenu. Left to this box, commands for selecting or deselecting all entries currently shown (labeled 'Check All' and Uncheck All', respectively) can be clicked. If more that one entry is exported to XML, these will be put into one XML file. An XML export containing multiple entries can be imported the same way as XML exports comprised of only one entry are. The testbed recognizes multi-entry exports and imports the individual entries properly. The format of multi-entry XML export files essentially is

the same as for single entry export files, so it is possible to separate multi-entry exports manually later.

It is helpful to use the testbed with more than one web browser windows or tab at once. Note however that in this case the 'Back' button of a browser will lead to the last page accessed by *any* open window or tab accessing the testbed. The effects of pressing the web browser's 'Back' button might be surprising and unintended from time to time.

3.3.3 Problem Types

The submenu for problem types, 'Problem Types', displays all problem types that are known by the testbed. Since they are not very numerous, no filters can be applied. Above the list of problem types, a selection box can be used to select a default problem type from the set of all problem types (see figure 3.2 on page 79). After selecting a problem type from the selection list, pressing button 'Set' will actually set the default problem type. Setting the default problem type works as a preliminary filter for the other submenus, filtering out all entries of a submenu that are not related to the default problem type, if applicable. Additionally, only information related to the default problem type is shown on many pages and a lot of settings are predefined automatically. This makes sense, because usually the user is only interested in an analysis for a single problem type at a time. All information related to a problem type is displayed in columns 'Problem Type' and 'Description'. The detailed view of problem types displays this information on a new page. Only the description of an entry can be edited and changed. The page for doing so looks like the page for creating a new problem type, only the text input field for entering a name is not editable. When creating a new problem type, however, the user must enter a name and an optional description (see figure 3.18).

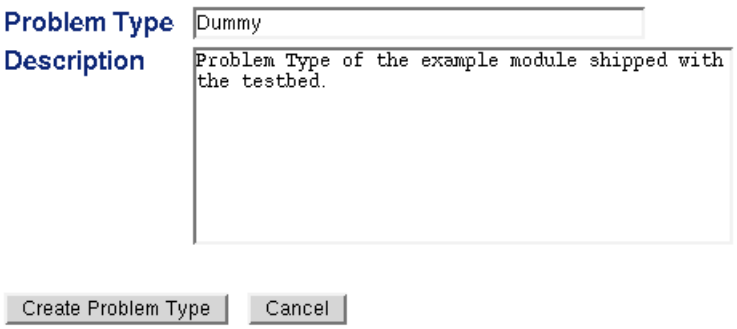


Figure 3.18: Creating a problem type

Deleting an entry entails that all data that is dependent on this problem type is removed from the testbed, too. That is, all configurations, modules, experiments, problem instances, jobs, and so on which belong to the deleted problem type are also automatically deleted from the testbed!

There is no special icon to export a problem type to XML, because the information about the problem type can easily be included in any XML export of a problem instance or algorithm by activating the export of the problem type in the user preferences (see subsection 3.3.12 on page 132).

3.3.4 Problem Instances

The submenu for problem instances, 'Problem Instances', displays the problem instances that have been imported into the testbed as discussed in subsection 3.2.3 on page 77 (see figure 3.17 on page 100) . If the default problem type is set, only problem instances of this type are displayed. The information shown for each entry consists of the problem type, a name, a description, and the date of import or creation of the instance. The textual data of a problem instance can be viewed by clicking on the 'Details' icon ( ).

Only the problem type and the description of an problem instance can be edited. To change the data comprising a problem instance, the user must import a new problem instance either by creating a new one with 'New' or by importing it via the command line interface (CLI) as discussed in section 3.4 on page 136. Note that no two problem instances with the same name can exist in the testbed. When creating a new problem instance with button 'New', a new page will appear. On the new page, the user can select the corresponding problem type with selection box labeled 'Problem Type', and enter a name and description via text input fields labeled 'Name', and 'Description', respectively. The contents of the new problem instance can not be input here since this is prohibitive because of the typical huge size of problem instance data. Instead, it is assumed that the contents is available as a file in the file system. By entering the file name including the correct path in text input field labeled 'File', the user can specify the file comprising the contents of the new problem instance. By pressing button 'Browse', the user can use a file browser to locate a file. Pressing button 'Create Problem Instance' will finally create the new problem instance, pressing button 'Cancel' will cancel the creation and leads back to the 'Problem Types' submenu.

3.3.5 Modules

There is a submenu for displaying modules integrated into the testbed, too (see figure 3.20 on the next page). This submenu is reached via link 'Modules' in the main menu.

Module parameters can not be edited via the web front end, only a module's description can be changed by using the according action. The page that will come up displays the problem type and the name of the module. Both information can not be changed. A text input field for entering or changing the description is labeled accordingly (see figure 3.21 on page 106). Changes can be submitted pressing button 'Change', the

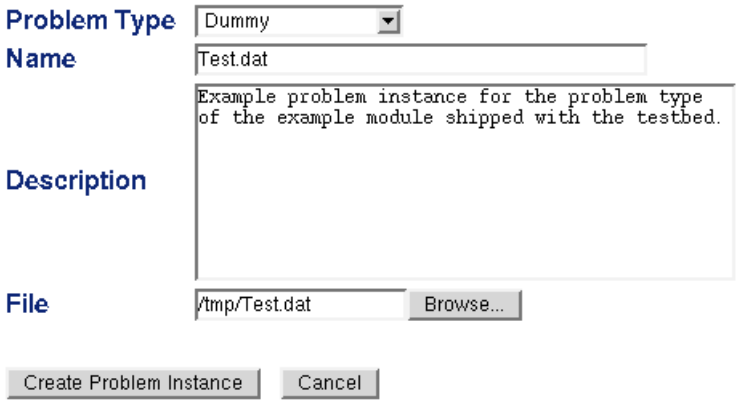


Figure 3.19: Creating a problem instance

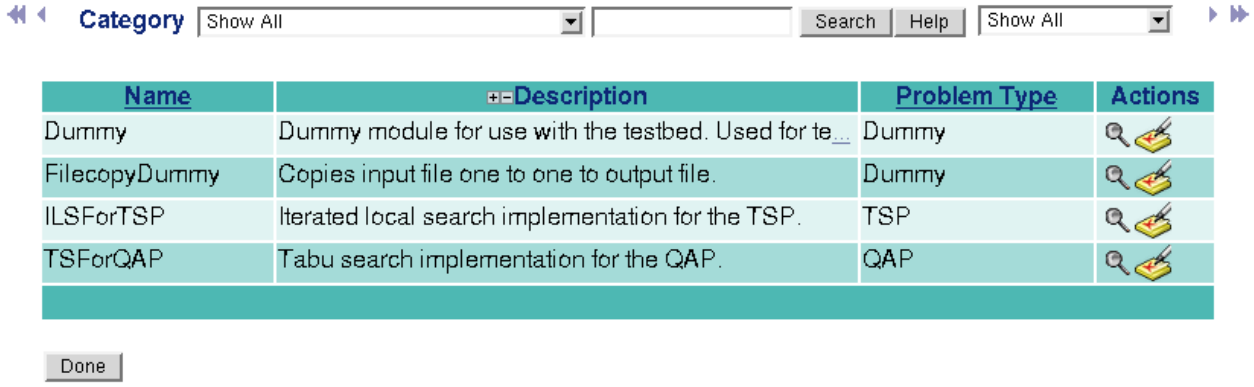


Figure 3.20: Modules submenu

whole operation can be aborted by pressing button 'Cancel' which will lead back to the 'Modules' submenu.

Note that any changes of the description will only affect the testbed database, the module definition file remains unchanged. Changing name or parameters settings of modules is only possible via the detour of deleting and re-registering the module under the same name again, however with the side effect that all dependent data of the deleted module will get lost (compare with the first part of subsection 3.3.2 on page 96). Modules are registered and incorporated into the testbed via the CLI (see subsection 3.4.2 on page 138). Creation of module definition files is described in detail in section 4.2 on page 181 and in subsection 2.3.1 on page 14.

Parameters and all other module data such as the description can be viewed using the 'Details' action ( ). The upcoming page will look like figure 3.22 on the following page.

The detailed view of modules presents the specification of the parameters of a module according to the module definition file: The columns contain the information that was exported by the module's command line interface definition output (compare to

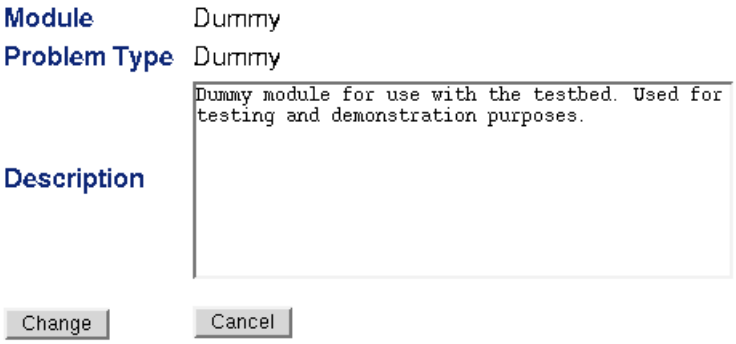
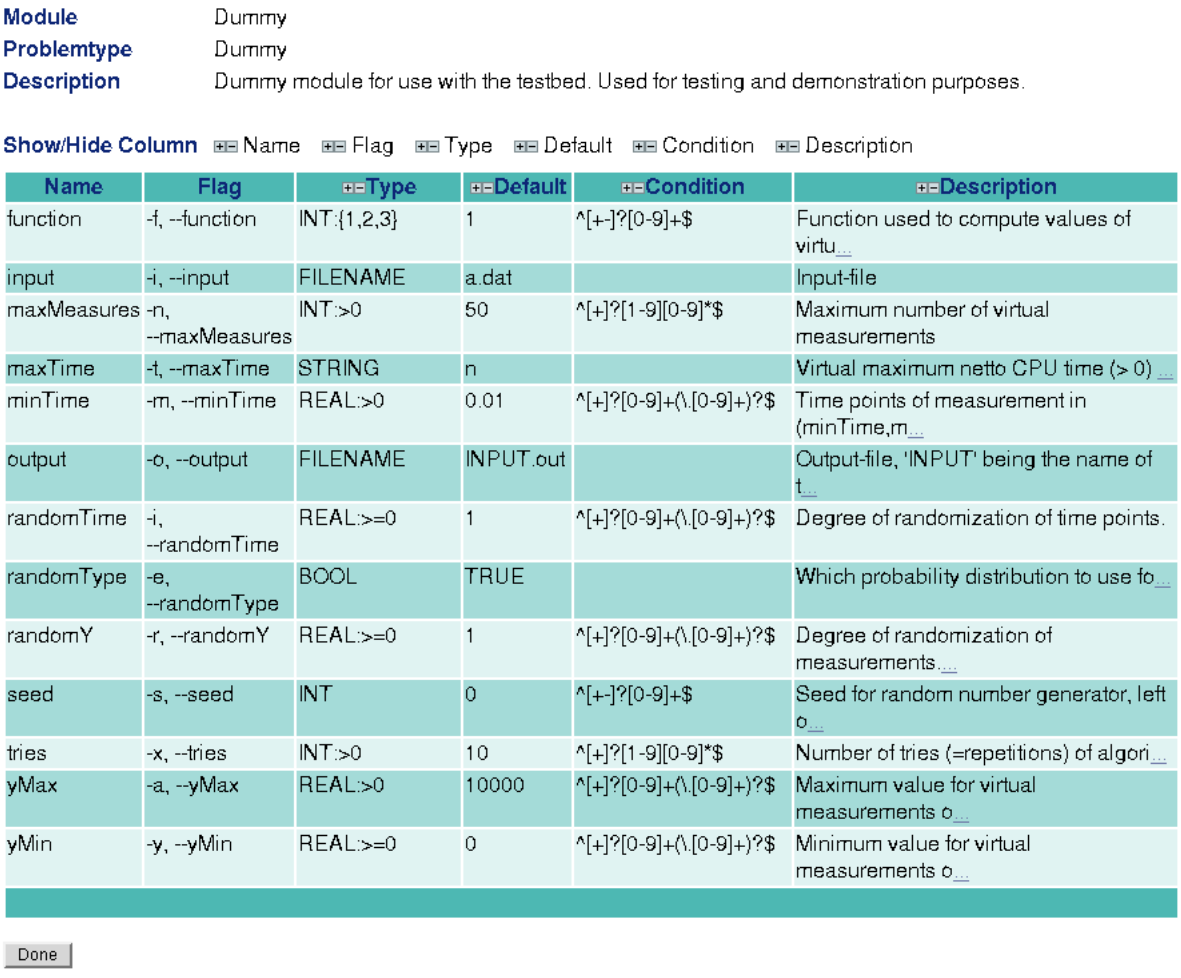


Figure 3.21: Edit description of a module



Name	Flag	Type	Default	Condition	Description
function	-f, --function	INT:{1,2,3}	1	^[+-]?[0-9]+$	Function used to compute values of virtu...
input	-i, --input	FILENAME	a.dat		Input-file
maxMeasures	-n, --maxMeasures	INT:>0	50	^[+]?[1-9][0-9]*$	Maximum number of virtual measurements
maxTime	-t, --maxTime	STRING	n		Virtual maximum netto CPU time (> 0) ...
minTime	-m, --minTime	REAL:>0	0.01	^[+]?[0-9]+(\.[0-9]+)?$	Time points of measurement in (minTime,m...
output	-o, --output	FILENAME	INPUT.out		Output-file, 'INPUT' being the name of t...
randomTime	-i, --randomTime	REAL:>=0	1	^[+]?[0-9]+(\.[0-9]+)?$	Degree of randomization of time points.
randomType	-e, --randomType	BOOL	TRUE		Which probability distribution to use fo...
randomY	-r, --randomY	REAL:>=0	1	^[+]?[0-9]+(\.[0-9]+)?$	Degree of randomization of measurements...
seed	-s, --seed	INT	0	^[+-]?[0-9]+$	Seed for random number generator, left o...
tries	-x, --tries	INT:>0	10	^[+]?[1-9][0-9]*$	Number of tries (=repetitions) of algori...
yMax	-a, --yMax	REAL:>0	10000	^[+]?[0-9]+(\.[0-9]+)?$	Maximum value for virtual measurements o...
yMin	-y, --yMin	REAL:>=0	0	^[+]?[0-9]+(\.[0-9]+)?$	Minimum value for virtual measurements o...

Figure 3.22: Detailed view of a module

the definition of the command line interface definition format in paragraph 'Parameter Specification of Modules' in subsection 2.3.1 on page 14). Each parameter is displayed in its own row. The name is displayed in the first column. Then, the long and short flags (column 'Flag'), next the type and subrange information (column 'Type') and a possible module internal default value (column 'Default') are given. The regular expression[11] in column 'Condition' is used to check any settings of parameters. Finally, the last column named 'Description' contains a description of the parameter.

The line labeled Show/Hide Column provides means to hide and redisplay columns to save space. This mechanism supports the expansion and implosion of columns to improve usability. The hide and show mechanism again works by pressing the + sign (⊞) to display a hidden column and by pressing the - sign (⊟) to hide a column completely (compare to the second part of subsection 3.3.2 on page 96).

Pressing button 'Done' in the detailed view of modules leads back to the last page visited. The detailed view of modules is also reached by clicking on a link in column 'Module Order' in the 'Algorithms' submenu, when viewing the details of an algorithm, when setting default parameters for an algorithm (see next subsection), or when configuring an algorithm (see subsection 3.3.7 on page 110).

3.3.6 Algorithms

This submenu, accessed via link 'Algorithms' in the main menu, lists all algorithms available in the testbed (see figure 3.23 on the following page). The submenu provides columns for presenting the name, description, problem type, and applicable operations for each entry. Column named 'Module Order' shows the number, identity and order of the modules the algorithm consists of. Further details about the individual modules can be accessed via clicking the underlined module names (compare with the last subsection). The detailed view of an algorithm lists all modules in the specified order, each module again listing its parameter specification and any default parameter value or any hiding specification as discussed later. Additionally name, problem type and description of an algorithm are presented. All this information can be accessed via editing the algorithm description, too. After creation or import of an algorithm, except for its description it can not be edited anymore, because other configurations and experiments might use this algorithm.

In order to create an algorithm with button 'New', on a new page (see figure 3.3 on page 79) the user must at least select the problem type from a selection box, type in a name for the algorithm and select one or more modules the algorithm is supposed to consists of. Depending on the problem type the modules available in the module selection box change. Since an algorithm can consist of more than one module, the user can get more module selection boxes for specifying subsequent modules by pressing

[11] as used in PHP, see Perl regular expressions in the PHP manual [54]

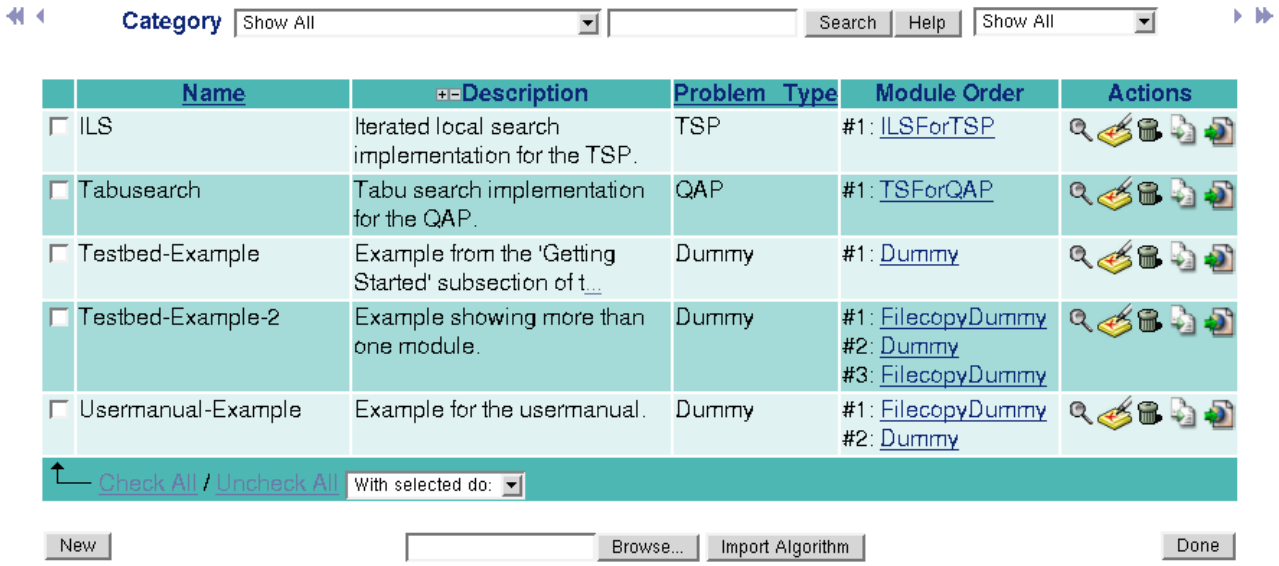


Figure 3.23: Algorithms submenu

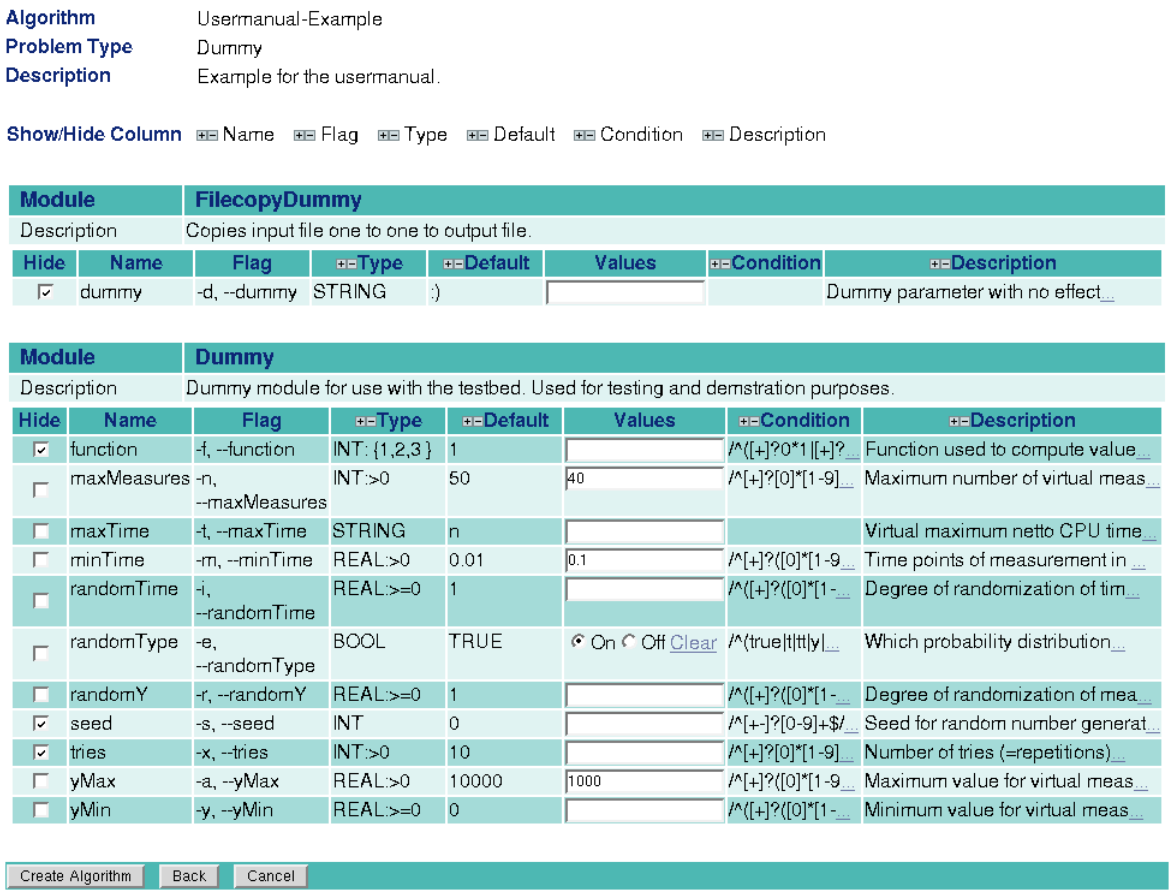


Figure 3.24: Creating an algorithm: Setting and hiding default parameters

button 'Module ++'. Pressing button 'Module --' will decrease the number of the module selection boxes by one and pressing 'Create Algorithm' finally will store the algorithm in the database. Button 'Cancel' will lead back to the 'Algorithms' submenu.

As discussed in subsection 2.3.1 on page 14 an algorithm is determined by the number, order and identity of its modules as well as the set of parameter that actually will be manipulable by the user when configuring an algorithm. Pressing button 'Set Parameters' leads the user to a new page where the user can fix parameter values for the algorithm (see figure 3.24 on the preceding page). The modules of an algorithm are presented in the order as specified on the previous page in the same way as they are presented in the detailed view of modules (see subsection 3.3.5 on page 104), only now column named 'Hide' to prevent a parameter from being configurable and colum 'Value' for entering parameter values are added. The functionality of hiding, expanding and imploding columns and single cells remains the same as for the detailed module view.

Parameters of a module of an algorithm can be set to fixed default value different from the module internal default value. Both can not be changed when configuring the algorithm later on. A parameter default value can be entered in the text input fields of column 'Values' for each parameter. Note that all real number have to be input in floating point notation. It is also possible to hide parameters to be subsequently invisible when configuring it or when extracting data from job results. Parameters that were fixed or hidden here can not be set later when creating a configuration. The user can hide a parameter by clicking on the check box in the first column. A hidden parameter is indicated by a mark as can be seen in figure 3.24 on the preceding page. Pressing button 'Create Algorithm' saves the algorithm with the parameters marked to be hidden and the fixed values for other parameters.

Hiding a parameter and setting it to a user defined default value different from the module internal default value is independent from each other. In particular, if a parameter is hidden and no further default value has been set by the user, the parameter will not show up when configuring the algorithm subsequently. When the module executable finally is called, the according parameter flag simply is omitted with the result that the module intern default value will be used. This entails that the parameter will not show up and hence is not accessible when extracting data from job results either (compare to subsection 3.3.10 on page 125 and section 4.3 on page 192). If the user additionally provides a default value for a hidden parameter, this default value will be used when the according module executable is called, i.e. the parameter flag will be used. A hidden parameter with a default value defined by the user will not be presented to the user when configuring an algorithm during the creation of a configuration. However, it will show up in the detailed view of a configuration (see subsection 3.3.7 on the following page) and it will be available when extracting data. If a parameter is not hidden and no default value is provided, the parameter can be configured later. Depending on whether a parameter actually has been configured or not, its flag will be used when calling the

according module executable or not, it will show up in the detailed configuration view or not, and it will be accessible in the data extraction results or not. If the user enters a default value for a parameter without hiding it, this parameter can not be configured any more but is still visible later on in the detailed view of a configuration and in the output of an data extraction effort, since the module executable is called with the according flag. If the user wishes to hide a parameter and simultaneously set the parameter to a default value other than the module internal default value in such a way that it is invisible in either the detailed view of an algorithm or configuration, when calling the executable, and in the data extraction results, it can be set in the internal parameters section of a module as is described in section a 4.2 on page 181. Note that the settings made in the internal parameters section are taken literally. If they are erroneous, e.g. the parameter name is misspelled, the executable might complain and fail to execute. Check the job server output to console for the exact call on system level.

In summary, hiding a parameter results in the testbed ignoring the parameter, while setting a parameter to a default value results in this parameter always showing up with the default value, the later having precedence.

Parameter Names

A module can be used more than once in an algorithm and a module can be used in more that one algorithm. In order to identify parameters with a unique name, the name of each parameter of an algorithm is build by concatenating the name of the module the parameter stems from, the position the module occupies in the algorithm, and the name of the parameter as exported by the module via its module definition file (coming from the module's command line interface definition output), separated by a '_'. As a result, it is guaranteed that each parameter in an algorithm has a unique name. These unique names can later be used to define conditions on different parameters when defining configurations for an algorithm (see next subsection). Note that the parameter names exported by a module can be changed in its module definition file (see subsection 4.2.3 on page 181). Even if the module exported a different name and even if a command line call will always use the long flag, the parameter's name in the testbed is as defined in the module definition file.

3.3.7 Configurations

Submenu 'Configurations' organizes the configurations contained in the testbed. Again, having set a default problem type provides a preliminary filter for all configurations in order to reduce the number of entries available for display. Entries can be deleted, but only the description can be edited, since other data might depend on a configuration such as an experiment using a configuration. Consequently, if a configuration is deleted, all

experiments that depend on that configuration are also deleted. The submenu for configurations features columns named 'Name', 'Problem Type', 'Description', and 'Action' with the typical meaning (see figure 3.25).

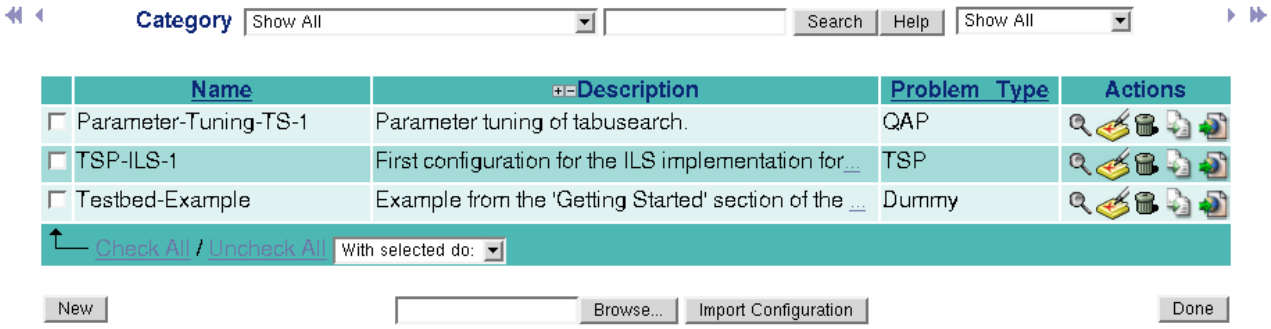


Figure 3.25: Configurations submenu

The creation of a configuration is split over three pages. A new configuration is created with button 'New'. On the first page (see figure 3.4 on page 80), at least the problem type, a name for the configuration and the algorithm that is to be configured must be selected. Depending on a problem type different algorithms will be available. By pressing 'Set Parameters' the next screen is presented which looks similar to the page for hiding parameters and setting them to a default values when creating algorithms as shown in figure 3.5 on page 81. On this page the user can enter the parameters for each module into text input fields as can be done for algorithm default values. The input fields in this case can contain values, *conditions* on values, loops of values and sets of values. These constructs can be used to define a broad range of possible value combinations of different parameters ranging from one single combination, i.e. one single fixed parameter setting to a full factorial design. Pressing button labeled 'Help' in headline of column 'Values' will pop up a separate window with information about syntax and semantics of the constructs for defining arbitrary conditions. Any parameter that was hidden will not show up, while any parameter that as attached an algorithm default value will present its default value without being changeable in a text input field. The topic is discussed in this subsection soon. Recall that a single value vector assigned to the parameters of an algorithm is called *a fixed parameter setting*. The notion of a configuration in this context is defined to be a set of such fixed parameter settings, compare with subsection 2.3.3 on page 32.

After each parameter for each module has been specified or left empty, in this case the module intern default value is assumed, the user presses button 'Submit Parameter Values' and is lead to the final page where individual fixed parameter settings can be saved and where the user finally has to give order to actually create the configuration after having seen the number and kinds all the resulting set of fixed parameter settings. This page (see figure 3.6 on page 82) is also reached, if an existing configuration is viewed in detail. On this page, the set of fixed parameter settings the configuration consists

of is shown as a table whose rows represent the individual fixed parameter settings and whose columns represent the individual parameters that have been set. On this page, it is possible to save certain single fixed parameter settings as a new configuration, by entering a name into the according text input field in the last column of an entry and pressing the 'Save as' button attached. This is useful, if for example during an experiment a special fixed parameter setting has attained a special status and will be reused afterwards. The whole configuration is finally saved by pressing the 'Create Configuration' button. The 'Back' buttons on the pages concerned with creating a configuration will lead back to the page visited last before starting the creation. The 'Back' button of the browser will cycle backwards through the pages involved by creation of a configuration.

Note that all real number have to be input in floating point notation. Additionally, filenames must contain path information when given through configurations of the testbed (see subsection paragraph 2.3.1 on page 15 and the according entry in the troubleshooting list in section 4.6), otherwise the files will not be found since it they are looked for in a temporary directory created by the testbed on demand.

In order to specify more than one fixed parameter setting for an algorithm, the user can enter several values per parameter input field as so called *sets* or *loops*. These values will, by default, be combined in each possible combination to form a set of fixed parameter settings. However, not all combinations obtained by this full factorial design or Cartesian product might be desired. The user can attach conditions to the sets and loops defined on the parameter values that altogether filter out some of all possible combinations of parameter values. The constructs set, loop and conditions will be described next,

Note: The back references between the pages displayed when a new configuration as created do not work properly. Most probably, using the browsers 'Back' button will work instead of the 'Back' buttons displayed on the pages.

Loops

If a the number of numerical values supposed to be input for a parameter is large and if this set can be constructed by some numerical construction scheme, the user can specify the values by using the loop construct. A loop has as similar syntax as a loop in programming languages. A loop is specified by entering a lower and an upper bound and a step size. A condition separated by a vertical bar can follow. A loop has the following format:

```
LB a ... b UB step c | COND
```

- `LB` is the lower bound qualifier of the loop. This can either be '[' or '(', indicating that the value for the lower bound is supposed to be contained in the set the loop represents or not, respectively. If no lower bound is specified '[' is used as default qualifier.

- `a` is a numerical value indicating the start or lower bound of the loop.
- `...` illustrates the loop.
- `b` is a numerical value indicating the end or upper bound of the loop.
- `UB` is the upper bound qualifier of the loop. This can either be ']' or ')'. Again, using the first qualifier indicates the end value will be contained in the set represented by the loop, while the second qualifier excludes the end value. If no upper bound is specified, ']' is used by default.
- `step c` defines the steps size of the loop. Value `c` must be a valid natural or real number greater than zero. If the step argument is omitted, `1` will be used as default.
- `COND` is a condition which is explained in the second next paragraph.

Sets

The user can enter more than one value in the input field for a parameter. An enumeration of several values is called a *set*. Elements of this enumeration or rather set are separated by commas. A set can contain only one type of data such as integers, reals or strings. Commas and backslashes can be escaped by a preceding second comma.

The format for a set is:

n_1, n_2, n_3 ... COND

The concluding condition restricts together with other conditions in the input fields of other parameters the number of possible combinations and will be explained now. Boolean parameters are treated differently, for the only sets that are possible for those is either a single `true` or `false`, or both. These can be selected via check boxes 'On', 'Off', and 'All', respectively. Clicking on link labeled 'Clear' of such an parameter will unset any settings which means to omit the according parameter in this configuration (see 3.5 on page 81 for an example).

Conditions

Conditions are used to eliminate some combinations of parameter values from the set of all combinations of parameter values, i.e. to eliminate some fixed parameter settings from the set of the full factorial design. Conditions work on a combination by combination basis. Typically, a combination of parameter values will be filtered out if at least one condition evaluated on basis of the values parameter of the combination evaluates to false. If one condition always evaluates to false for all parameter combinations values, the set of combinations will be empty.

Conditions work based on the names of parameters and constants such as numbers and strings. These names are variables representing the actual value of the parameter with this name in each combination. Conditions will be evaluated by PHP. Hence, all functions and operators from PHP (such as `==`, `!=`, `<=`, `<`, and so on), even regular expressions, can be used for the definition of a condition. The unique names of the parameters (compare with naming convention of parameters of an algorithm in paragraph 'Parameter Names' on page 110 in subsection 3.3.7) are used to declare necessary dependencies between two or more parameters. Two types of conditions are available in the testbed. They are visualized by `|` and `|*`. The first type of condition,`|`, works, as described before by first building a full factorial design of parameter value and subsequent elimination of all combinations for which at least one conditions attached to a parameter fails. The second type of condition, `|*`, called *relaxed condition*, is a relaxed version of the first type of condition. The second type of condition works by restricting the building of the full factorial design in advance. In this case, if such a condition is attached to a parameter, the set of values as defined by a loop or set construct is only used, if the condition evaluates to true. That ways, it is possible, for example, to use parameters that are only available if another parameter has a certain value. If normal conditions are used, each combination of the full factorial design has the same amount of parameters in a combination. If relaxed conditions are used, the amount may vary.

Note that the equality in PHP is expressed by `==` (equal sign twice). The single equal sign is used in PHP to assign values to variables and, if used in a condition, evaluates to true.

In order to illustrate the use of sets, loops, and conditions, some examples are depicted next.

Example 1

Parameter Name	Input
`Dummy_1_maxMeasures`	`5...30 step 5`
`Dummy_1_maxTime`	`10,12,20 | Dummy_1_maxTime == Dummy_1_maxMeasures`

The condition will only be true for the two combinations of parameter values (10,10) and (20,20). Consequently, only 2 out of 18 possible combinations from the full factorial design ($\{5, 10, 15, 20, 25, 30\} \times \{10, 12, 20\}$) will form the configuration.

Example 2

Parameter Name	Input
`Dummy_1_maxTime`	`5...30 step 5`
`Dummy_1_minTime`	`(5...30) step 5 | Dummy_1_maxTime > Dummy_1_minTime`

All combinations (x, y) with x representing the value for parameter `Dummy_1_maxTime` and y representing the value for parameter `Dummy_1_minTime` where $x \geq y$ will be filtered out. Hence, the configuration will comprise only 10 combinations out of 24 (6×4) of the full factorial design which would result from dropping the condition.

Combination	1	2	3	4	5	6	7	8	9	10
`Dummy_1_minTime`	5	5	5	5	10	10	10	15	15	20
`Dummy_1_maxTime`	10	15	20	25	15	20	25	20	25	25

Example 3

Parameter Name	Input
`Dummy_1_maxTime`	`1,2,3`
`Dummy_1_minTime`	`5,6,7 |* Dummy_1_maxMeasures == 8`
`Dummy_1_maxMeasures`	`7...9 |* Dummy_1_minTime == 3`

This configuration will result in the following parameter combinations ('-' indicates that the parameter will not be used for a combination):

Combination	1	2	3	4	5	6	7
`Dummy_1_maxTime`	1	2	3	3	3	3	3
`Dummy_1_maxMeasures`	-	-	7	8	8	8	9
`Dummy_1_minTime`	-	-	-	5	6	7	-

3.3.8 Experiments

This submenu displays the data representing experiments, one entry per experiment (see figure 3.26).

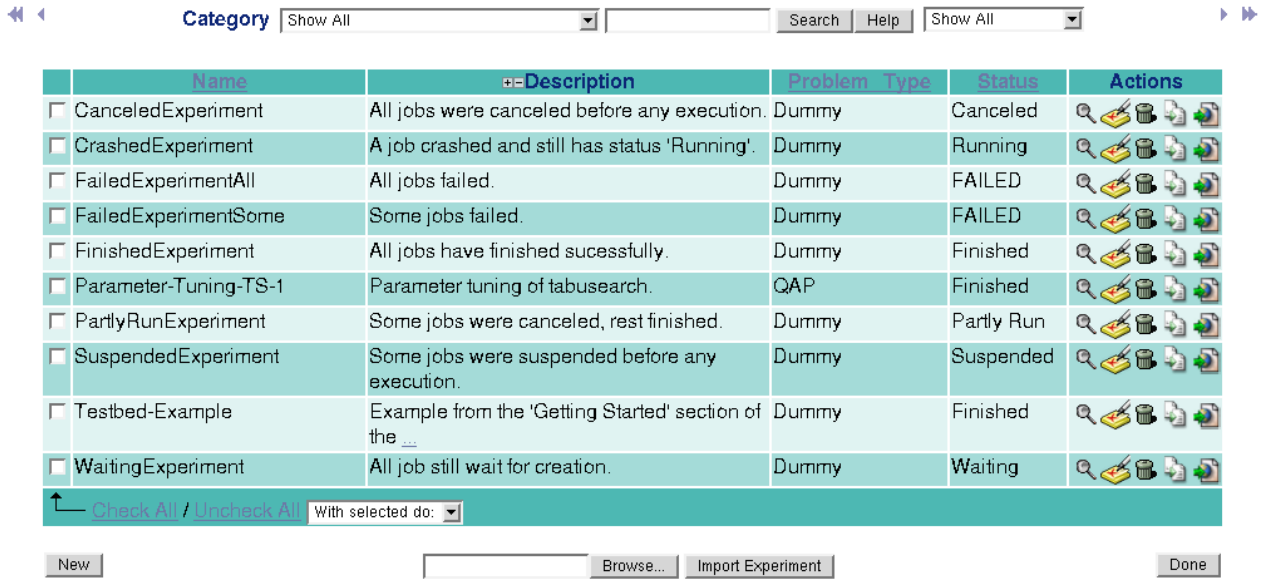


Figure 3.26: Experiments submenu

Filters work as in the submenus described before. As is the case for configurations and for the same reasons only experiment descriptions can be edited. The columns are similar to those in the 'Configurations' menu except for column 'Status'. Column labeled 'Status' shows for each entry the current status of the experiment. How this status is determined is described in table 3.3 on the facing page.

A new experiment can be created with the 'New' button. A new page will come up (see figure 3.7 on page 83). On this page the problem type must be selected. Depending on the problem type, the configurations and problem instances selectable in the selection boxes below will change. Next, a name and at least one problem instance and configuration must be chosen. Multiple problem instances and configurations can be selected by holding down key 'Control/Ctrl' while clicking on the corresponding entries in the selection boxes. If check box 'Store Job output to console in database' is selected, the output of the processes that are executed when running a job of the experiment to standard output, i.e. what would normally be printed to the console, is stored in the database, too. In the 'Jobs' submenu, the user can see this output by clicking on the output icon (, see table 3.6 on page 124). Depending on the module this output can become very huge, so this check box is not activated by default. Pressing button 'Create Experiment' stores the experiment to the database. However, no jobs are created or started at this moment. After creation of an experiment this way, the user is automatically lead to the next page of the experiment creation procedure where the user can view which jobs will

result from the experiment specification (see figure 3.8 on page 86). This upcoming page is essentially the same as the detailed view page for an experiment that can be reached by pressing the 'Details' icon (🔍) on the 'Experiments' submenu for an entry. Pressing 'Done' will not start any jobs but will set the experiment to status 'Waiting' and will lead back to the 'Experiments' submenu. Via the detailed view of an experiment, it can be started later. By pressing button 'Start Experiment', the jobs will be created and put to the job execution queue.

Experiment Status	Definition
Waiting	If the experiment has been specified but not started yet, no jobs have been created an hence have not been put to the job execution queue, yet. In this case, the status of the experiment is set to 'Waiting'.
Running	If at least one job is still running or waiting to be run, i.e. has status 'Running' or 'Waiting', respectively, the whole experiment status will be 'Running'.
Finished	If *all* jobs have been run properly and now *all* have status 'Finished', the experiment status is set to be 'Finished', too.
Suspended	If at least one job has been suspended with status 'Suspended' while no job is still running with status 'Running' or waiting to be started with status 'Waiting', the experiment status becomes 'Suspended'.
Canceled	If *all* jobs have been canceled and set to status 'Canceled' before any of them had the chance to run on the system, the experiment is considered to be canceled with status 'Canceled'.
FAILED	If all jobs have been run or canceled (statuses 'Finished', 'FAILED', or 'Canceled', respectively), while at least one job has failed with status 'FAILED', the experiment status entailed is 'FAILED', too.
Partly Run	If all jobs have been run properly yielding status 'Finished' or have been canceled yielding status 'Canceled' with each status occurring at least once in an experiment, the experiment status as a whole is 'Partly Run'.

Table 3.3: Experiment statuses

The detailed view of an experiment presents all information about an experiment such as its name, its status, its description, a list of all configurations and problem instances

employed, and an overview over all jobs that result from the experiment settings. The details page will list all jobs an experiment consists of in the form of a table. Each job occupies one row. The columns illustrate the number of a job, its fixed parameter setting, the problem instance used, its status, and the actions that can be performed on a job (the actions featured for a job are considered in the next subsection about jobs). Recall that it is possible to use several configuration based on different algorithms with different parameter names in the same experiment. If in fact multiple configurations are used which configure different sets of parameters of possibly different algorithms, columns of parameters not used in a fixed parameter setting for a job will be left empty for the according job.

The status of an experiment is determined by the statuses of the job that belong to the experiment. A job can have six different statuses. These job statuses are listed in table 3.5 on page 123. The status of an experiment is defined based on the statuses of its jobs as is summarized in table 3.3 on the page before.

In short, the status of an experiment will be 'Waiting' as long as no job has been created yet. if jobs have been created, the status of an experiment will be 'Running' as long as at least one jobs is still running or waiting. If this does not apply, an experiment status remains 'Suspended', if at least one job remains to be suspended. Now assume, all jobs of an experiment have finished execution either successfully, by failing, or by cancellation. In this situation, the experiment status will be 'FAILED' as soon as at least one job failed. If *all* jobs have been canceled, status 'Canceled' will occur. If all jobs of an experiment except at least one finished successfully while at least one job has been canceled, the experiment status will be 'Partly Run'. Finally, if and only if *all* jobs have finished successfully, the experiment status will become 'Finished'.

Depending on the status of an experiment, different buttons at the end of the detailed view page can be used. They will trigger actions on the jobs of the experiment and hence will change job and experiment statuses as is explained next:

- If an experiment has status 'Waiting', it can be started with button 'Start Experiment'. This will create the jobs according to the experiment specification and will set all job statuses to 'Waiting' and accordingly the status of the whole experiment to 'Running'.
- If the experiment has status 'Running', button 'Suspend' can be used to suspend all waiting jobs of an experiment. All running jobs or all jobs that have finished execution in one form or the other (statuses 'Finished', 'FAILED', or 'Canceled') remain unaffected. As soon as the last job of the experiment finishes execution, the status of the experiment changes to 'Suspended', if at least one job actually has been suspended, or to status 'Finished', 'FAILED', or 'Canceled' according to the definition of experiment status based on its job statuses, if then all jobs have finished execution.

An experiment with status 'Running' can be canceled with button 'Cancel'. This will cancel all waiting and suspended jobs. Jobs with other statuses are unaffected. The experiment status will change to 'Canceled', if all jobs were still waiting. It will change to 'Partly Run' as soon as at least one job has finished successfully and all the other jobs were still waiting. If some jobs had statuses other than 'Waiting' or 'Suspended', the new experiment status will eventually be 'Finished', 'FAILED', or 'Canceled' depending on the final jobs statuses as soon as all already running jobs have finished execution.

- If the experiment status is 'Suspended', button 'Resume' can be used to resume all suspended jobs of the experiment. All other jobs remain unaffected. The status of the experiment will change to 'Running'.

  Status 'Suspended' features button 'Cancel', too, which will cancel all suspended jobs, leaving the experiment with status 'Canceled', 'Partly Run', or 'FAILED'.

- Experiments with status 'Canceled', 'Partly Run' or 'FAILED' can be restarted with buttons 'Restart' and 'Retry', respectively. This will set the status of all jobs to 'Waiting', the status of the experiment changes to 'Running' as a consequence. The restart counter of all jobs is incremented.

For each job in the detailed view page of an experiment, the second last column shows the current status of the job. Clicking on the hyperlink for a job in column 'Status' will present a page with the output of the job and the specification of the job. If the job is currently running this output might be incomplete. Note that the job output accessible here is the contents of the file the last module of an algorithm outputs to the file named by the value of parameter flag `--output`.

If the experiment was just created, all jobs will wait to be started. In this case the experiment can be started by pressing the 'Start Experiment' button as was explained before. In case of this first start, the user can also assign a priority to the jobs of the experiment. Jobs with the highest priority will be executed first by a job server. By default, the priority is implicitly set to 50. Additionally, the user can specify on which hardware the jobs should run on. If the testbed is used in a network of computers, the computational power of machines in the network might differ. In the default setting, 'On same Hardware', the jobs will be run on computers in the network that have been identified with the same alias. Which alias or rather class of equivalent computers are used is determined by the machine the first job of the experiment was started on (see 3.3.14 on page 134 for more information about hardware aliases). With the setting 'Any hardware', the jobs may be run on any machine of the network possibly having quite different computing power. This might execute some jobs of the experiment faster than others since they are distributed over differently powerful computers.

Be aware that results of an experiment may be useless if the algorithms of the experiment are not designed to produce result which are independent from computational power of

Experiment Testbed-Example
Problem Type Dummy
Status Finished
Description Example from the 'Getting Started' section of the testbed manual.

Configurations **Problem Instances**
Testbed-Example 100.Dummy.dat

Created Jobs

Parameters							
JobNo	Dummy_1_maxTime	Dummy_1_maxMeasures	Dummy_1_randomY	Dummy_1_yMin	Problem Instance	Status	Action
978	110	30	1.5	1	100.Dummy.dat	Finished	
979	110	30	1.5	1.2	100.Dummy.dat	Finished	
980	110	30	1.5	1.5	100.Dummy.dat	Finished	
981	110	30	1.5	2	100.Dummy.dat	Finished	
982	110	30	1.5	2.5	100.Dummy.dat	Finished	
983	110	30	2	1	100.Dummy.dat	Finished	
984	110	30	2	1.2	100.Dummy.dat	Finished	
985	110	30	2	1.5	100.Dummy.dat	Finished	
986	110	30	2	2	100.Dummy.dat	Finished	
987	110	30	2	2.5	100.Dummy.dat	Finished	
988	110	30	3	1	100.Dummy.dat	Finished	
989	110	30	3	1.2	100.Dummy.dat	Finished	
990	110	30	3	1.5	100.Dummy.dat	Finished	
991	110	30	3	2	100.Dummy.dat	Finished	
992	110	30	3	2.5	100.Dummy.dat	Finished	
993	110	30	4	1	100.Dummy.dat	Finished	
994	110	30	4	1.2	100.Dummy.dat	Finished	
995	110	30	4	1.5	100.Dummy.dat	Finished	
996	110	30	4	2	100.Dummy.dat	Finished	
997	110	30	4	2.5	100.Dummy.dat	Finished	

Restart Experiment | Clear Job Console Output

Figure 3.27: Detailed view of an experiment

the machines they are run on. Other settings by selecting a specific alias will distribute the jobs on several machines, too (namely the machine belonging to that alias). In this case, these machines were deemed some kind of comparable by the user, otherwise they would not have been assigned the same alias. See subsection 3.3.14 on page 134 for more information about defining aliases.

Note: A job server is started by entering `testbed server` on the CLI (compare to section 3.4 on page 136).

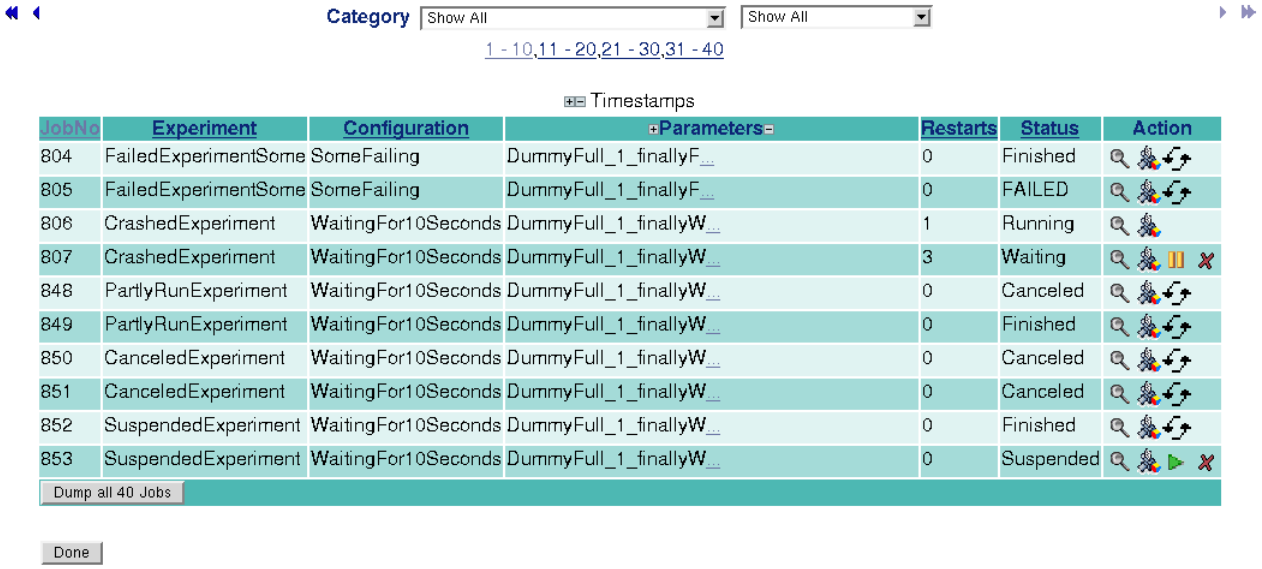


Figure 3.28: Jobs submenu

3.3.9 Jobs

The submenu for jobs presents the jobs administrated by the testbed (see figure 3.28). It features category and experiment but no regular expression or problem type filters[12].

The table listing the jobs available for display shows columns quite different to columns of other submenus. These columns are named 'JobNo', 'Experiment', 'Configuration', 'Parameters', 'Generated', 'Started', 'Ended', 'Restarts', Status', and 'Actions'. These columns provide information about an entry's job number, the experiment and configuration it features, its parameter settings, the point in time it was generated, the point in time it was started for the last (re-)start, the point in time it ended its last execution, the number of restarts, its status, and the actions applicable depending on its status, respectively. Recall from subsection 2.3.3 on page 32 that jobs are identified using a unique integer job number. The 'Jobs' submenu has another noteworthiness compared to other submenus for it provides a means to expand and implode all columns related to the presentation of timestamps at once by clicking on the +/− sign (±) labeled 'Timestamps' as is shown in figure 3.28 (compare to subsection 3.3.5 on page 104).

The actions that can be performed with entries of the 'Jobs' submenu differ from those of other submenus. They are essentially dependent on the status a job has. In table 3.5 on page 123 the different statuses a job can have together with a short definition is presented. Table 3.6 on page 124 presents the actions available and their effects together with their representing icons. Table 3.4 on the next page finally summarizes the applicable action

[12]By creating a special category only querying for a specific problem type, a problem type filer can easily be constructed. Compare to section 3.5 on page 146 and specifically to subsection 3.5.1 and 3.5.2 on pages 146 and 165, respectively.

Status	Applicable Actions	New Status	Sideffects
Waiting for Creation	None	Waiting after having been created.	
Waiting	Suspend	Suspended	Job virtually removed from job execution queue.
	Cancel	Canceled	Job removed from job execution queue.
Running	None	Running	Job processes are running on the system.
Finished	Restart	Waiting	Job put to job execution queue. Restart counter incremented.
Suspended	Resume	Waiting	Job virtually put to job execution queue again.
	Cancel	Canceled	Job finally removed from job execution queue.
Canceled	Restart	Waiting	Restart counter incremented.
FAILED	Retry	Waiting	Restart counter incremented.

Table 3.4: Actions application to jobs

with respect to the job statuses. The effect of the actions is given, too.

Note that jobs can not be edited or created by hand. Jobs are created automatically by specifying an experiment and always belong to this experiment. Jobs can not be deleted manually. They are removed automatically, if the experiment they belong to is deleted. Note also that once a job actually has been started, i.e. the processes of its module executables have been actually started on the system, it is no longer in the job execution queue. Any such job can not change its status until its processes have finished running. Hence, a job crash is only detected by recognizing that the job's status is 'Running' for more than a specific amount of time. This amount of time can be defined in the config file `TESTBED_ROOT`*/config.php* (see paragraph 'Starting a Job Server' on page 140 in subsection 3.1.4 on page 69). This, however, can in some cases not be detected

Job Status	Definition
Waiting for Creation	This status is virtually only, since the job exists only virtually. The job's experiment has been created, the jobs of the experiment, however, have not yet been created nor put to the job execution queue. Such jobs will not show up in 'Jobs' submenu, but only in the detailed view of the experiment, until the experiment they belong to has been started. In this case, the jobs will be created, stored to the database, and put to the job execution queue by setting their statuses status 'Waiting'.
Waiting	The job has been created, stored to the database and put to the job execution queue by setting it to this status. The job is waiting and prepared to be executed. In essence, a job is in the job execution queue if and only if its status is waiting.
Running	The job's processes actually are running on the system.
Finished	The job's processes have been run properly on the system without encountering any errors and have finished with an exit code indicating success. The testbed was able to store the job output back to the database.
Suspended	The job has been put to the job execution queue but before its processes could be started it has been ordered not to be executed until further notice, i.e. until the 'Resume' action is applied to it.
Canceled	The job has been put to the execution queue, but before it could be started, it was canceled by removing it from the job execution queue and setting it to this status. The job can be restarted, though.
FAILED	The job's processes have been run on the system but encountered problem so they exited with an exit code indicating failure, or, alternatively, the testbed was not able to store the job output back to the database, or was not able to provide the input file (problem instance), or was not able to execute all module executables of the job successfully for some other reason.
No such Job	This is not a real job status. It indicates in the detailed view of an experiment that something is wrong with either the experiment specification or the database state. This can happen for example, if an XML import went wrong. Compare to subsection 3.4.3 on page 139.

Table 3.5: Job statuses

Icon	Action	Effect
	Restart/ Retry	Each job can be restarted after it has run or waited for execution once. Whether the job actually was run or not or whether the run was successful or not or is irrelevant: The action can be applied to any job having status 'Finished', 'FAILED', or 'Canceled'. The restart counter for the job is increased by one, the new status of the job is 'Waiting' and hence is put to the job execution queue. Note that any old output to standard output or the final result stored in the database will be overwritten (compare with subsection 3.3.8 on page 116.
	Suspend	If a job has not been executed yet and still is waiting for execution in the job execution queue, i.e. has status 'Waiting', it can be suspended and later be resumed. The status of a suspended job will become 'Suspended'. The job virtually remains in the job execution queue but a special status is preventing its execution.
	Resume	A suspended job can be resumed by canceling its special status that prevents it from being executed. This is done be setting the job to new status 'Waiting'. As a consequence, the job is put back to the job execution queue and can and will be executed as soon as the job is first.
	Cancel	If a job has not been executed yet (status 'Waiting') or if it is suspended (status 'Suspended'), it can be marked as canceled with status 'Canceled'. The job will not be executed by the testbed, because it is removed from the job execution queue immediately.
	Show Stdout	This action shows the output of the job's processes to standard output, i.e. to the console. A new page will open which shows this output (which is not the job's result output as specified via parameter `--output`). If the job is still running, the output might be incomplete, but it will be updated regularly during execution. The output will only be available if the experiment was started with check box 'Store Job output to console in database' activated when creating the job's experiment (compare with subsection 3.3.8 on page 116). Note that any restart will overwrite old output .

Table 3.6: Actions for jobs: Icons and effects

automatically, but has to be triggered from time to time by command `testbed reset` on the CLI. This command will, among other things, set the status of all running jobs to status 'FAILED', hence they can be restarted. See section 3.4.5 on page 142 for more information about this topic.

3.3.10 Data Extraction

As explained in subsection 2.3.1 on page 31, an algorithm consists of one or more modules that are executed sequentially. Each module in the sequence expects its input via an input file and writes its output to an output file. Information about these files is transmitted to modules by means of command line parameters. The final output of an algorithm, besides any error or status messages will be a file.

After an algorithm with a given fixed parameter setting has been executed on a specific problem instance, i.e. after a job has been run, the contents of the last output file, i.e. the result of the job, is stored in the testbed database. The output file format should be conform to the standard output format of the testbed as described in subsection 2.3.1 on page 24. This is because, if the user wishes to undertake a statistical analysis, these output files can be further processed easily within the testbed in order to serve as input for a statistical tool, in the case of the testbed the R package.

Data extraction scripts (or extraction scripts for short) are used to extract data from the result of jobs. This, however, can only work automatically , if they can rely on some basic format of the output regardless which algorithm produced it. Extraction scripts scan the results, extract certain information and provide them as tables of data in a way similar to tables in relational databases. In this table form the data extracted can easily be conveyed to and processed by the R package.

Extraction scripts are managed via the submenu 'Scripts' in submenu 'Data Extraction' (see figure 3.29 on page 126). How to write extraction scripts is explained in detail in section 4.3 on page 192. The scripts are listed in a table with columns displaying the name, description, and contents of a script. As with entries in other submenus, extraction scripts can be edited, deleted, exported to XML, or imported from XML. Import works by choosing an XML file representing an extraction script with the file browser facility of the web browser by pressing 'Browse' and then 'Import Script' as was described in the second part of subsection 3.3.2 on page 96. Since data extraction scripts generally are not related to either a specific problem type or an experiment, experiment or problem type filter do not apply here. Regular expression filter work on the name, description, and contents of the scripts. The usage of category filters does not yet apply here, although a selection box for applying category filter is provided. The reason is that categories can not be build for scripts yet.

A new script is created with button 'New Script' in the 'Scripts' submenu. On the upcoming page a name, a description, and the script itself can be entered in the accord-

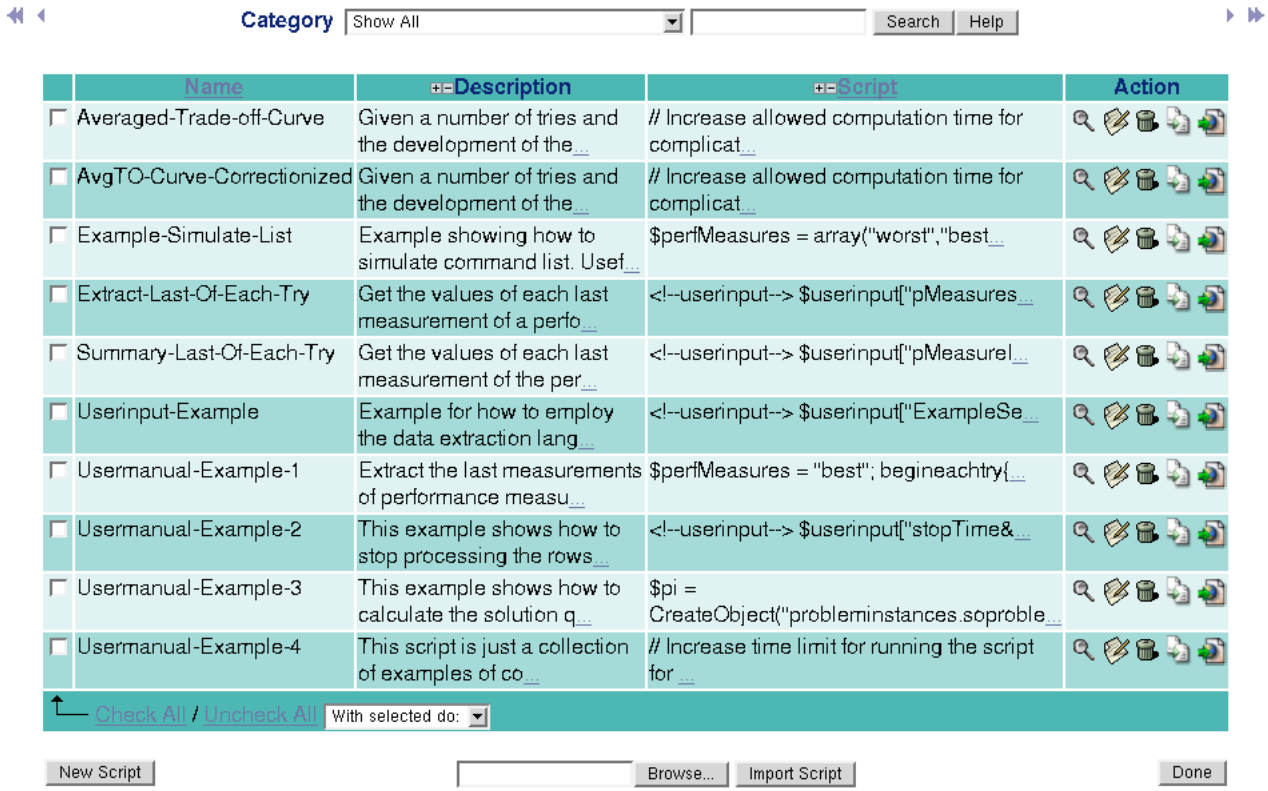


Figure 3.29: Data extraction script submenu

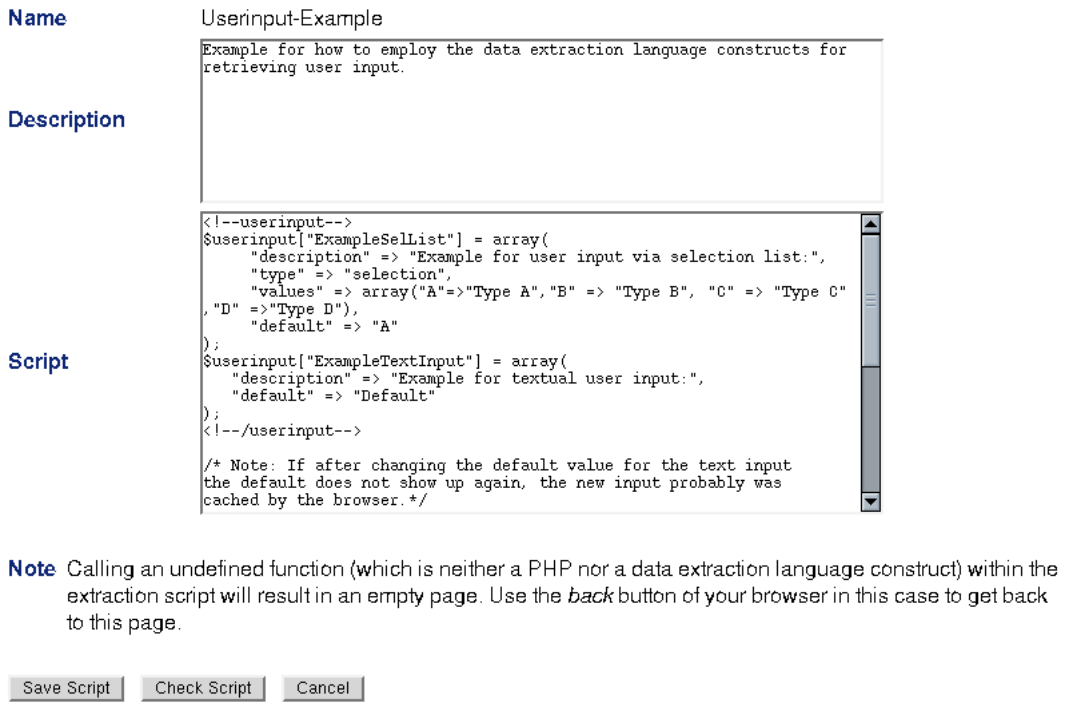

Name
Userinput-Example
Example for how to employ the data extraction language constructs for retrieving user input.
Description
<!--userinput-->
$userinput["ExampleSelList"] = array(
"description" => "Example for user input via selection list:",
"type" => "selection",
"values" => array("A"=>"Type A","B" => "Type B", "C" => "Type C"
,"D" =>"Type D"),
"default" => "A"
);
$userinput["ExampleTextInput"] = array(
"description" => "Example for textual user input:",
"default" => "Default"
);
<!--/userinput-->
/* Note: If after changing the default value for the text input
the default does not show up again, the new input probably was
cached by the browser.*/
Script
Note Calling an undefined function (which is neither a PHP nor a data extraction language construct) within the extraction script will result in an empty page. Use the back button of your browser in this case to get back to this page.
Save Script
Check Script
Cancel

Figure 3.30: Creating a data extraction script

ingly labeled text input fields (see figure 3.30 on the preceding page). The size of these input fields can be changed in the 'Preferences' submenu (see subsection 3.3.12 on page 132). Button 'Check Script' on this page can be used to verify the script for syntax errors. Depending on the severity of the error, the upcoming page may be incomplete or completely empty. A typical error, for example, is the call of a function inside the script which does not exist. In this and all other error cases, the user is encouraged to use the 'Back' button of the browser and check the previously entered script for undefined function calls. Note that empty scripts are not allowed. A simple empty comment `//` will do, however. Button 'Save Script' can be used to finally store a newly entered or changed old script to the database. Button 'Cancel' leads back to the 'Scripts' submenu for data extraction scripts.

Some useful generic extraction and analysis scripts have been exported to XML and are located in directory `DOC_DIR`*/scripts*. Data extraction scripts are always named **.X.xml* while analysis scripts are named **.R.xml*. Example and other generic data extraction and analysis scripts illustrating some aspects of writing scripts, e.g. user input, are located in `DOC_DIR`*/scripts/analysis* and `DOC_DIR`*/scripts/extraction*, respectively. To import all scripts available in these two directories, import files *Standard-All.R.xml* and *Standard-All.R.xml*, respectively, since these contain all other scripts in multi export format.

Extraction scripts are applied in submenu 'Data Extraction' (see figure 3.11 on page 88). When data is to be extracted from a set of jobs or rather the set of job results, the user first selects an extraction script to be used from the list of all data extraction scripts available as listed in selection box named 'Extraction Script'. Next, the user specifies the set of jobs on which results the script will work on from selection box 'Experiment'. The user can choose an experiment from the list of all experiments in the testbed or an arbitrary set of jobs that is defined by the current search filter or a category (information about search filters and categories can be found in subsection 3.5 on page 146). All these means to define to define sets of jobs are selectable here.

If the extraction script chosen needs additional input from the user, new input fields under the caption 'Input requested by employed Data Extraction Script' will be shown at the end of the page where the user can enter the information required. The user input for extraction and analysis scripts is determined through the script that is used. For data extraction scripts as well as for analysis scripts, special language constructs for requesting user input exist. These input requests can be of two types. Either they are of textual nature or the user is supposed to select from a number of given choices. The first kind of user input is handled by a text input field. A possible default value will be already entered and can be changed by the user. The latter kind of user input is entered via a selection box with the default selection already selected. Additional information given by the author of the scripts concerning the nature and purpose of the input requested will be written left to the input request. For more information

about user input requests for scripts compare with sections 4.3 on page 192 and 4.4 on page 212.

After having specified everything to extract data, the user has to declare what to do with this data. Six choices are available:

1. 'View Result in HTML'
2. 'Download as CSV (comma separated)'
3. 'Download as CSV (tabular separated)'
4. 'Download as HTML table'
5. 'Download as LaTeXtable'
6. 'Analyze with R'

The data extracted can be viewed as a HTML table directly on a new page directly (1.), can be saved to disk (2. - 5.), or can be directly redirected to and used by an analysis script. The suited action can be selected with the corresponding check button.

Viewing the result of an extraction process on a new page (1.) can be used to check the results of an extraction script application, e.g. to test a new data extraction script or to get an overview of some experimental results. Note that the amount of data produced by an extraction effort usually is quite huge. Loading this data into the web browser as is done when using action 1. can take a while. Actions 2. - 5. are used to store the data extracted to disk in various formats. CSV files are files representing a table of data as a list of lines, each line containing one entry, columns of entries separated by commas (2.) or tabulator (3.). The CSV format is widely known and can be read by the R statistics package for example. Download as HTML table (4.) will produce the same results as when viewing the result in HTML on a new page (1.) except that in this case table definition as plain HTML code can be stored directly to disk. The download of any data extracted as a latex table also produces the same results as the other download facilities, only the table produced is encoded as a LaTeXtable. The latter two downloads are intended for further text processing of data extraction results. In any case, after selecting the desired download option, a file browser will open to let the user determine where to store the resulting file and under which name.

If the data extracted is to be analyzed directly with an analysis script, option 'Analyze with R' (6.) has to be selected. Currently, only analysis scripts in the form of R scripts are supported. The analysis script to be used can be chosen from selection box 'Analysis Script' which displays all analysis scripts available in the testbed. If the analysis script chosen needs additional user input, the requested information can also be entered on the page under caption 'Input requested by employed Analysis Script'.

Before actually starting the extraction and any subsequent processes, the user can specify which columns to appear in the resulting table representing the data extraction results. As is explained later in detail in subsection 4.3.1 on page 193, each piece of data is represented as one single line in a table. The columns of the table represent the single components or attributes of each piece of information or rather the atomic measurements. These components typically comprise which job and hence experiment, configuration, algorithm, problem instance, and on on produced it and the settings of the parameters of the algorithm that produced the measurement. Not all columns are needed and since tables can become quite big it is desirable to be able to prune superfluous columns of result tables. Exactly this can be accomplished by pressing button 'Calculate Columns'. Pressing this button starts a dummy extraction process. No data will actually be extracted, but the appearance of the would be result table is computed. The columns of the table will be displayed in a selection list. The user now can highlight all columns to appear in the result table when really starting the data extract with button 'Extract Data' by clicking the corresponding column names in the selection list. Again (compare with subsection 3.3.8 on page 116), holding down key 'Control/Ctrl' while clicking an entry will add the according column to the set of highlighted and hence displayed columns. Of course, this selection procedure can be skipped if the result table need not be changed. After pressing 'Extract Data', the output of the analysis script will be shown or the file browser of the web browser will open. Note that analysis scripts used by the 'Data Extraction' submenu must produce textual output, if the output is to be displayed by the testbed. Handling graphical output of analysis scripts is explained in the next subsection. The analysis scripts for plotting shipped with the testbed will output their status and error messages in textual form here.

3.3.11 Statistical Analysis

Scripts for the statistical analysis called analysis scripts are managed in the 'Scripts' submenu of submenu 'Data Analysis' (see figure 3.31 on the next page). All analysis scripts available are presented as a list with the same structure as the list in submenu 'Scripts' of submenu 'Data Extraction'. The available actions and filters are the same as in that submenu, too: Entries can be edited, deleted, exported to XML or imported from XML. New analysis scripts can be added with 'New Script'. On the upcoming page a name, a description, and the script itself can be entered (see figure 3.32 on the following page). The settings of the text input field sizes for entering data extraction scripts apply here, too.

If any extracted data has been saved as a comma separated CSV file, this data can be used as input for an analysis script. This is done in submenu 'Data Analysis' (see figure 3.14 on page 93). In fact, if the script is supposed to produce graphical output, e.g. plots, it can not be applied in the 'Data Extraction' submenu (see figure 3.11 on page 88) concerned with data extraction as described in the previous section. Instead,

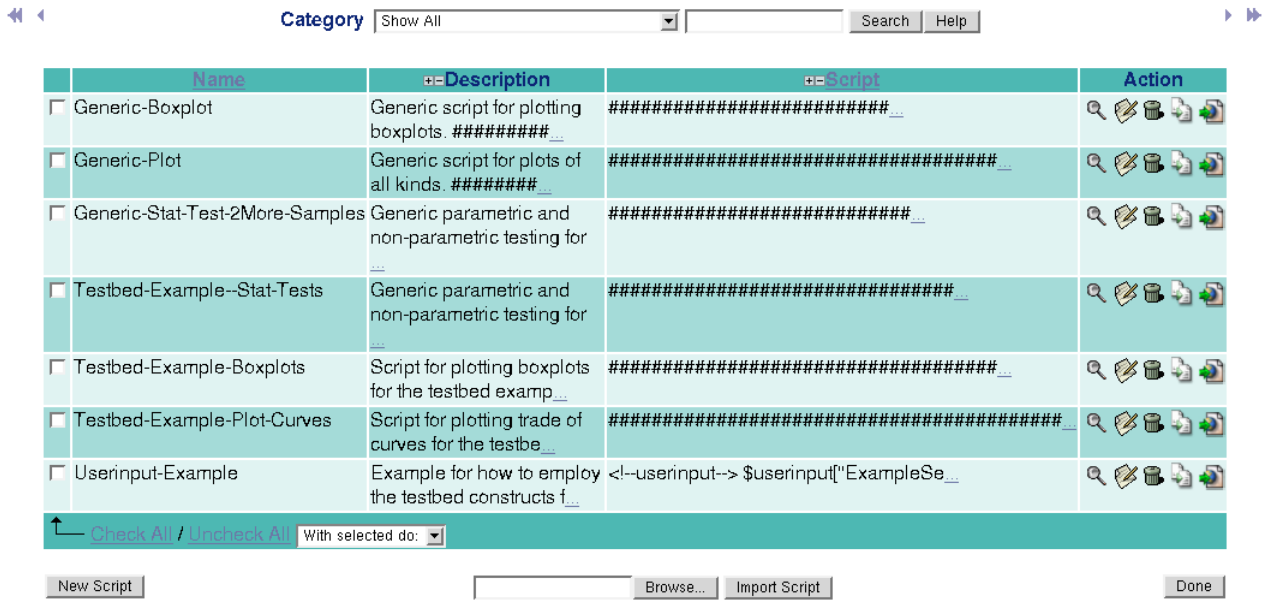


Figure 3.31: Analysis scripts submenu

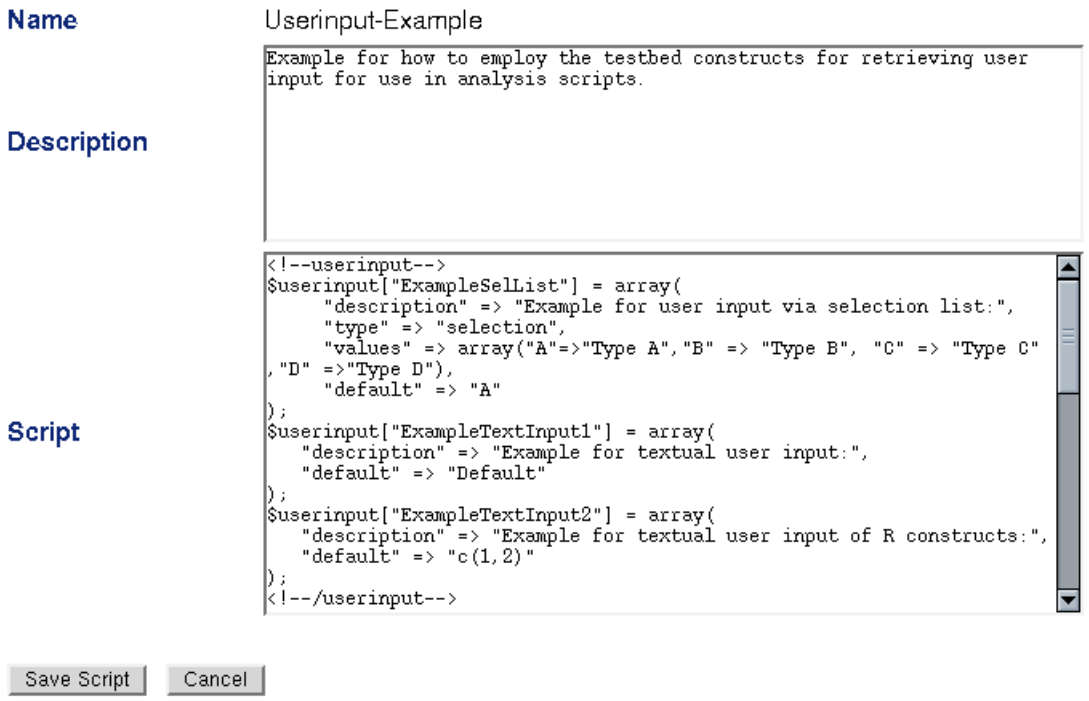

Name
Userinput-Example
Description
Example for how to employ the testbed constructs for retrieving user input for use in analysis scripts.
Script
<!--userinput-->
$userinput["ExampleSelList"] = array(
"description" => "Example for user input via selection list:",
"type" => "selection",
"values" => array("A"=>"Type A","B" => "Type B", "C" => "Type C","D" =>"Type D"),
"default" => "A"
);
$userinput["ExampleTextInput1"] = array(
"description" => "Example for textual user input:",
"default" => "Default"
);
$userinput["ExampleTextInput2"] = array(
"description" => "Example for textual user input of R constructs:",
"default" => "c(1,2)"
);
<!--/userinput-->
Save Script
Cancel

Figure 3.32: Creating an analysis script

it must be used by means of the 'Data Analysis' submenu. To apply an analysis script in this submenu, first the analysis script is chosen from the list of all analysis scripts available as contained in the selection box named 'Analysis Script'. Next, the CSV file containing the data is selected via the file browser that shows up after pressing button 'Browse' which is located behind label 'Data File'. Before actually starting the analysis with button 'Start Analysis', the user can specify whether the files and scripts generated during the analysis should be kept on the testbed server for later download. This is indicated by clicking check box 'Keep Files'. So it is possible to generate images via R and store them in the temporary testbed directory. All generated images, the data used for the analysis and the analyze script itself can be downloaded. It is possible to repeat and prove the results of an analysis. If the check box is not selected, any files produced by the analysis script are discarded and only the scripts textual output will be available on an a new page.

Again, if the analysis script needs additional information, this information is requested with additional input fields that will show up after a script has been selected. The type of input fields available her are the same as for data extraction scripts (see previous subsection). After entering any requested information, the analysis is continued by pressing button 'Start Analysis'.

If option 'Keep Files' was activated, a link to download the analysis results and generated images as a compressed `tar` archive (ending *.tgz* will be available after the analysis has run (compare to 3.16 on page 94 and 3.16 on page 94). Those files will automatically be deleted after 5 days. For each analysis a new link is generated and those links are not listed anywhere again, so the user must download the files when the page is shown or save the link to download the files later. For more information about creating plots within the testbed, see subsection 3.2.8 on page 85.

Analyzing data directly from within the testbed with analysis script unfortunately has some restrictions. The only graphical file format supported by the testbed is that of bitmaps in the *.png* format. Additionally, errors occurring during the run of an analysis script can not always be transferred with the proper error messages to the testbed. Finally, it is quite cumbersome to develop and adjust an analysis script exclusively from within the testbed, since running an analysis script always involves loading the data into R before execution of the script. For these and possibly further reasons, sometimes it is desirable to use the R package standalone. The procedure, however is straightforward. The analysis scripts as stored in the testbed can be used directly with copy and paste via the clipboard, only the line for reading in the data input has to be modified slightly (see section 4.4 on page 212). Any results of data extraction effort stored as CSV files can be loaded by R as is, no additional reformatting is needed. For further information about issues related to the statistical analysis with the help of analysis scripts, see section 4.4 on page 212 and the documentation for R [60].

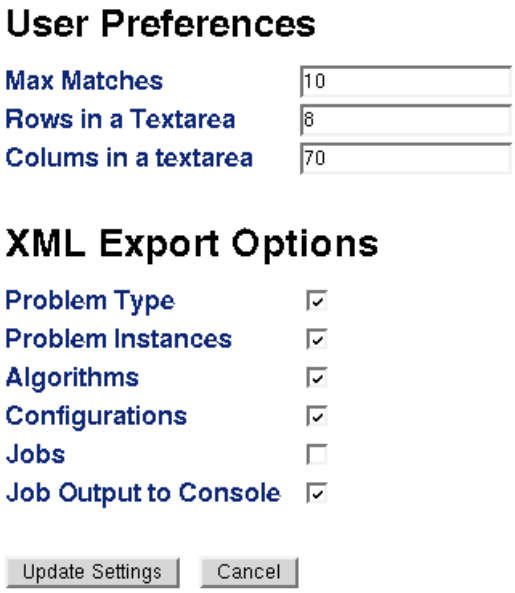


Figure 3.33: Preferences

3.3.12 Preferences

In the 'Preferences' submenu several global settings can be adjusted. Often, if entries from submenus are exported to XML they depend on other information in the testbed (compare with the second part of subsection 3.3.2 on page 96). For example, an experiment contains links to the configurations and problem instances used, while the configurations in turn contain links to the algorithm they configure, and so on. Now, when such an experiment is exported with the aim to import it again, the configuration objects it depends on must either be exported and subsequently imported, too, or a links in the form of names are exported that correspond to the configurations the experiment uses. The latter case assumes that the configuration objects the links represent are already present in the testbed when the experiment is imported again. In case of a data transfer from one testbed to another, the latter usually is not the case. In the XML export options of submenu 'Preferences' the user can specify which dependencies will be adhered during XML export by exporting other required objects automatically to XML, too. Objects an exported entry is dependent on (like the problem instances used in an experiment) will only be exported if the check box for that group of objects is marked (see figure 3.33). The XML representation of the additionally exported objects is printed into the same file as the entry that is exported originally. For more information about the subtleties involved by ex- and import via XML see the details in subsection 3.4.3 on page 139.

By entering a natural number in the input field for 'Max Matches', the user can specify how many entries from the set of entries available in each submenu will be displayed on one page, i.e. the segment size (compare with part one of subsection 3.3.2 on page 96).

Additionally the size of large text input fields when editing a description or a data extraction or analysis script can be defined in terms of number of rows and columns. The declaration refers to any large text input field used within the testbed.

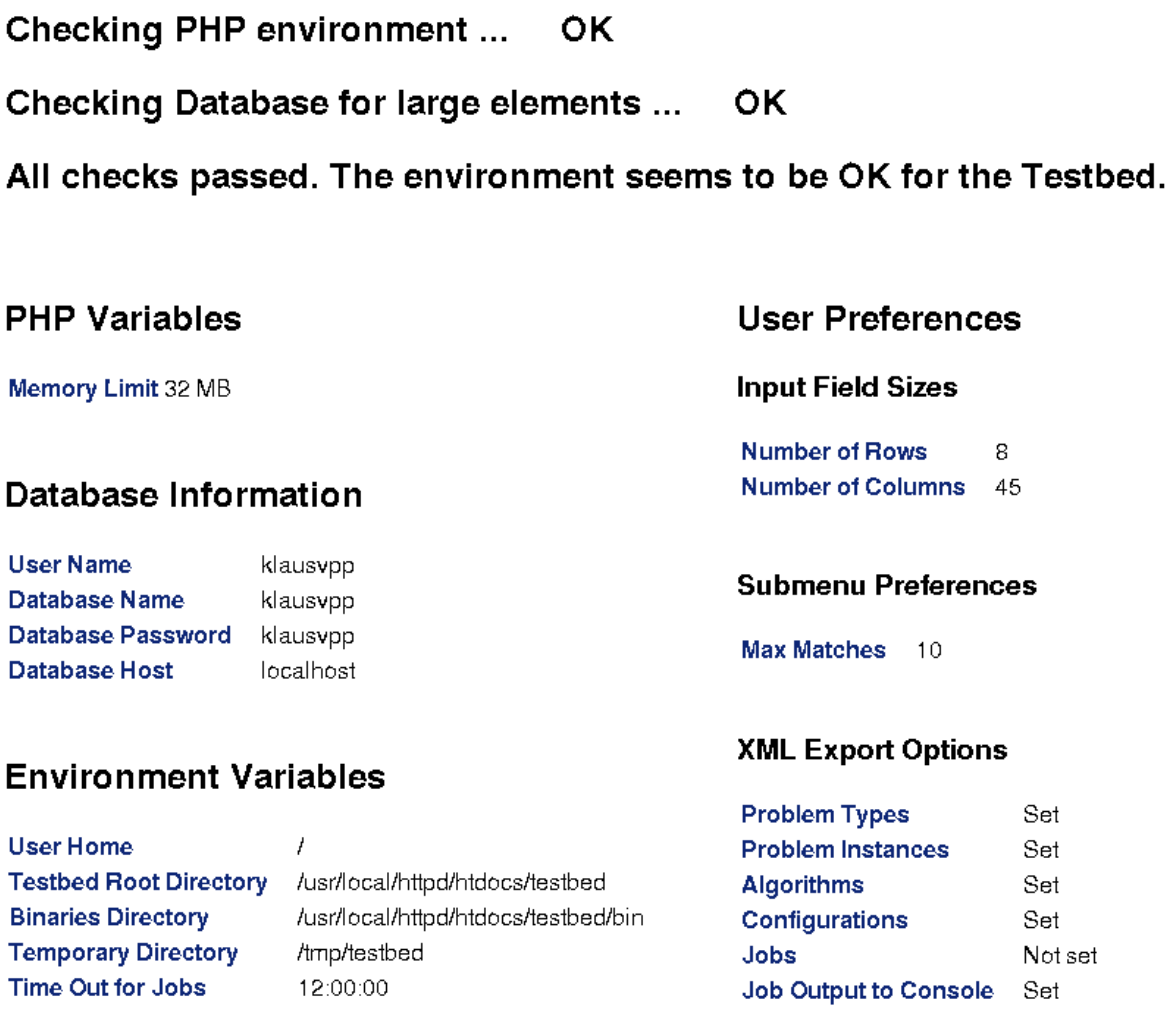


Figure 3.34: Testbed status submenu

3.3.13 Testbed Status

Submenu 'Testbed Status' provides a page that checks for the status of the testbed and displays the results it nicely structured (see 3.34). It checks various settings that are required for the testbed to run smoothly such as whether the PHP version is up to date, whether magic quotes are disabled, and so on (compare to section 3.1 on page 51). Furthermore, it checks and displays the current memory limit for the testbed and possibly proposed an increase. Other information such as the name of the testbed database, the current database user name and password, certain environment variables,

and the current user preferences are showed also. If some settings of the testbed prevent that the checks can be done, the submenu page might not be accessible. In these case, link 'Check' below the link for submenu 'Testbed Status' can be accessed. This link is *http://localhost/testbed/check.php* (*localhost* denotes the host of the testbed web server).

3.3.14 Hardware Classes

Each time a testbed job server is started on a client machine in a network of computers with a central testbed server, information about the CPU of the machine the job server was started on is stored in the database. Any job server started by a user knows by means of the user's configuration file were the central testbed server and hence database is located. (compare to section 2.5 on page 42 and section 3.1 on page 51). In a network of computers, based on such CPU identifiers, the user can create *hardware aliases* which represent classes of machines with equivalent computational power in the network. For example, a mobile PIII 1Ghz PC has the same computing power as a desktop PIII 1Ghz PC. However, they may have different CPU identifiers. By default, only identical CPU identifier are treated as being computationally equivalent. Hence, it is necessary to provide means to identify two CPU with different name as being computationally equivalent. In submenu 'Hardware Classes' of submenu 'Preferences', all different CPU identifiers found in the network by the testbed are displayed in a list (see figure 3.35 on the next page). For each entry, the user can enter an alias in the input field in the last column. The aliases define classes of equivalent hardware, i.e. of CPUs which are deemed to be computationally equivalent. After entering the alias of a hardware class in the text input field, the alias for the entry is stored by pressing button 'Set'. With button 'Delete' the CPU identifier is removed from the database. The hardware classes defined by the aliases can be used when starting an experiment (see subsection 3.3.8 on page 116): Aliases or CPU identifiers are used to designate on which machine(s) an experiment's jobs are supposed to run on. As mentioned before, a job server will retrieve information about the CPU it is running on and compare this information with the aliases in the database. After that, it will fetch any waiting jobs present in the database which are designated for the alias of the CPU identifier of the machine it runs on. The order is not be determinable in advance. Equivalently, it can not predicted, which job server will fetch which job from the database. An alias is detached from an CPU identifier by submitting an empty alias on the according entry in the list of hardware classes with 'Set'.

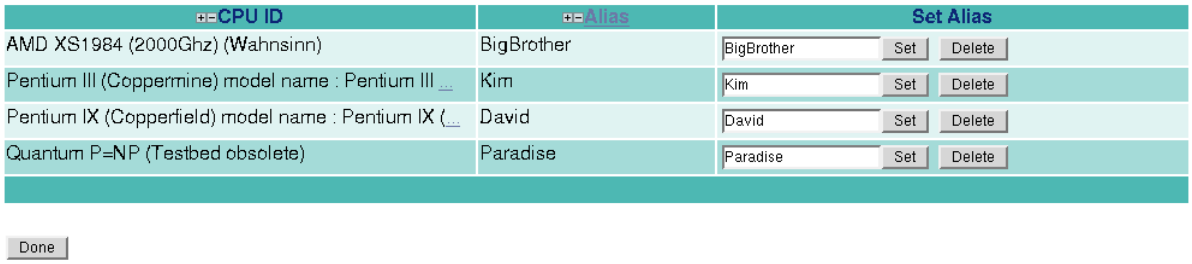


Figure 3.35: Hardware classes

3.4 Command Line Interface (CLI)

The testbed user interface is web based. In principle, any parts of it are accessible via the Internet. However, some tasks such as starting and running a job server and running the jobs itself must eventually be performed on a local (client) machine. Since it is not desirable to have full access to a local machine via Internet, some processes and tasks must be started directly on the local machine. This subsection describes the tasks that can be accomplished locally using the command line interface (CLI) which is the testbed's user interface to be used with a shell or console. In order to enable this, a command line tool, i.e. a stand-alone PHP program, has been developed. It can perform all local tasks by calling it with appropriate arguments just as any other system tool. The following list presents all possible main arguments or sections together with a short explanation. The different main arguments are described in the subsequent subsections individually. The main arguments or rather sections of the testbed command line tool are:

Extract	Extract data from results of Jobs
ProblemInstances	Im- and export of Problem Instances
Modules	Add and remove modules
Import	Import previously exported XML data
Server	Start a job server
Backup	Backup testbed
Restore	Restore a previously backuped testbed
Vacuum	Vacuum the testbed database
Reset	Reset/clean up the testbed database
Jobs	Display Job output
Dump	Display general data structure of testbed objects

To get more information about an individual section use command

```
testbed help <section>
```

3.4.1 Extract Data

Data from result files of jobs which have been run outside the testbed can be extracted using the testbed data extraction scripts, too, without having to rerun the jobs within the testbed or without having to import the output files into the testbed. Such external result files can be extracted by the testbed, if the output complies to the standard output format of the testbed as defined in subsection 2.3.1 on page 24. Any data

extraction scripts contained in the testbed can be used for extracting these external results. They will put the data extraction outcome in the table format used by the testbed (see subsection 4.3.1 on page 193) in the form of a comma separated list, as well. Using command

```
testbed extract -l <scriptname>
```

the user can see which user input in the form of fillable variables for the data extraction script named `<scriptname>` is expected. These variables can then be set on the CLI with `-v <varname>=<value>` (`<varname>` denotes the name of a variable). The general command line call is defined as (`<file 1>`, ..., `<file n>` are the result file to extract from):

```
testbed extract [-v "varname=value"]*[13] <scriptname> <file 1> ... <file n>
```

A complete command can look like:

```
testbed analyze -v pMeasuresInput=best Extract-Last-Of-Each-Try
                                                    <file 1> ... <file n>
```

The output is a comma separated list with a header line naming the columns of the table represented by the list. The format is the same that is produced when exporting data with the 'Download as CSV (comma separated)' option in submenu 'Data Extraction' (see subsection 3.3.10 on page 125). The CSV files thus produced can then be further processed by external statistics packages, for example the R package since R can read in CSV files (compare with subsection 3.3.11 on page 129). For more information about type and purpose of expected user input see the description of the data extraction script in the web interface of the testbed. Note that the parameters that show up as columns stem from the begin/end parameters section of each output file in contrast to the any data extraction effort started via the web based user interface. The parameters used in the latter case stem from the values that were used to run the jobs. These are not available in stand-alone result files, though, so only parameters and their values stored in a result file can be used.

Problem Instances

Problem instances can be added to and extracted from the testbed via the CLI, too (compare to subsection 3.3.4 on page 104). A Problem instance stored in the testbed can be retrieved with command:

```
testbed probleminstance get <name>
```

The content of problem instance named `<name>` will be printed to standard output and can be redirected to a file: The user can send the contents directly to another program or write the content to a file by redirecting the standard output to this file by adding

[13] A '*' means zero or more occurrences of the bracket it follows.

'> <filename>' to the command mentioned before or by using the piping mechanism (|).

It is possible to add problem instances that have been exported to XML with the web interface. However, the user can import and create only one problem instance at once this way. Using the CLI the user can use wildcards to add many problem instances at the same time. This is done with command:

`testbed probleminstance add <problem type> <file 1> ... <file n>`

If problem type `<problem type>` does not exist in the testbed, it will be created automatically, however, without any description. The problem type can be edited in the testbed in submenu 'Problem Types' (see subsection 3.3.3 on page 103).

3.4.2 Module Management

Modules are used to build algorithms. They can be managed, i.e. added or removed only via the CLI. Changing the description of module, however, can be done in the web user interface, too[14] (see subsection 3.3.5 on page 104. The module management options include registering a new module as is described in subsection 3.2.2 on page 76 and removal of a module. In order to register a module, command

`testbed module register <modulename>`

is used. The module can only be registered, if the module definition file for the module that is to be registered is in its correct location in the file system, which is `TESTBED_BIN_ROOT/` *modules*. If the problem type of the module does not yet exists in the testbed it is automatically created, however, without any description, which can be added later, though. The module's executable must be in its correct location, too, as is described in subsection 3.2.2 on page 76. A module can only be removed from the testbed if no algorithm uses it. In order to remove a module from the testbed, all algorithms, configurations and experiments that use the module must be removed first via the web front end. A module can be removed with command

`testbed module remove <modulename>`.

If there is still an algorithm in the testbed that uses the module, a warning will be shown and consequently the module will not be removed. Note that module names may only consist of characters 'a' - 'z', 'A' - 'Z', and '0' - '9'. Special characters such as '-', '_', or ',' are not allowed in module names and will be removed silently when automatically generating a module definition file (compare to section 4.2 on page 181 and subsection 2.3.1 on page 14).

To generate a module definition for module (binary) `<modulename>` file according to the

[14]Note that when changing the description of a module with the help of the web user interface of the testbed, this change is not automatically stored in the according module definition file, either.

command line interface definition format of the testbed, the following command can be used:

```
testbed modules makeConform ./<module>
```

in the directory the module binary is located. To generate a module definition file for a module with non-conform a command line interface, issue:

```
testbed modules makeNonConform ./<module>
```

The two commands

```
testbed modules makeConform
```

and

```
testbed modules makeNonConform
```

simply are aliases for the tools that actually do generate the module definition file. They can be found in directory `TESTBED_ROOT`*/devel/* and are named *gen_module_from_mhs.php* and *gen_module.php*, respectively (compare to subsection 4.2.1 on page 181).

3.4.3 Importing Data

Almost any variable data from the testbed can be exported to XML. There are two different ways to import data that was exported back into the testbed. Small files (less than 1MB) can be imported directly with the web front end. Bigger files should be imported using the CLI. XML filescan be imported on the CLI with command

```
testbed import <XML file 1> <XML file 2> .. <XML file n>
```

The reason why it is advisable to import bigger file only via the CLI is that while importing data the memory consumption can get very big. PHP restricts the amount of memory a PHP application can use. If the files to be imported are very big, the PHP memory limit must be high enough. By default, the testbed CLI runs with 128MB of maximum memory. The memory consumption mainly depends on the size of the data imported. If integral parts of an XML file to be imported have a huge size, for example 10MB, the memory consumption importing this part can be much higher, in case of the example up to 50MB. If the XML file contains only a lot of small elements, it is possible to import files that are even bigger than the memory limit. Generally, if more than 128MB of memory is needed for the testbed, shell script `testbed` (located in */usr/local/bin/* or */usr/bin/* has to be edited manually: Number `128` in parameter `memory_limit=128MB` has to be substituted by a larger number. During an import effort, some tables of the testbed database may get locked because the transaction of importing data is in progress. This can lead to a slow responding web front end.

The import of XML files come with some subtleties. When exporting an object contained in the testbed, the user can choose in the 'Preferences' submenu (see subsection 3.3.12

on page 132) whether to export certain other objects automatically. These other objects are characterized by the fact that the object that originally is to be exported depends on them. Objects that are exported automatically this way are called secondary objects, the object that originally is to be exported is called primary object. Primary objects depend on their secondary objects. The alternative to automatically export secondary objects, too, is to export just links in the form of names.

Now, when importing any exported objects, it is necessary to either import any secondary objects, too, or to make sure these are already contained in the testbed. If the secondary objects were exported, too, they can be imported, too, if no other object with the same *type and name* already exists in the testbed. In the latter case, the testbed assumes that a secondary object already exists and hence does not import it. A warning will be issued and the import of the other secondary and the primary objects will continue and perhaps successfully end. If no secondary objects were exported, the testbed checks, whether objects of same *type and name* do exist in the testbed already. If the do not exist, the import will fail. Otherwise, it may continue.

This procedure unfortunately has some disadvantages. If the name of an object contained in the testbed and a secondary object of an import effort that is to be imported are identical, but in fact refer to two different objects, the database will be semantically in an inconsistent state which entail strange errors and behavior. The user has to make sure that identical names really refer to identical objects, otherwise XML export and import does not work. If in doubt, always export any secondary objects completely, too and check names before re-importing any data. If name clashes occur, rename the objects affected manually in the XML file. The XML file is easy to read, so this should not be a problem.

3.4.4 Starting a Job Server

When an experiment is created, no processes or rather jobs will be started automatically. Instead, when an experiment has been started by the user, the resulting jobs are added to the job execution queue by setting their statuses to 'Waiting'. The job execution queue is governed by all job servers that are running somewhere in the network and that are connected to the testbed database collectively. Each job server periodically accesses the database and picks waiting jobs, i.e. jobs from the job execution queue, and executes them. If no job server is running or connected to the database anywhere in the network, no jobs can be executed. The jobs with the highest priority will be executed first. Otherwise, an ordering of jobs is not possible (compare to sections 2.5 and 3.1 on pages 42 and 51, respectively).

If jobs are assigned to an equivalence class of computers, a job server can only execute jobs of an experiment, if the computer the job server runs on belongs to the assigned equivalence class. Recall that the user can specify special hardware requirements and

classes of equivalent computers found in the network in the form of aliases for the jobs to run on when defining an experiment (compare with subsection 3.3.14 on page 134). The default is 19.

A testbed job server is started with the following command on the CLI:

```
testbed server [<options>]
```

The options available are (`<int>` denotes an integer value):

- `-n <int>` or `--nice <int>`: Run the module executables with which nice level `<int>` in the system (see man page for command system nice [72]).
- `-m <int>` or `--max-memory <int>`: Do not use more than `<int>` MB of memory when running a module binary. The default value is the amount of physical memory installed on a machine. Use -1 to use all available memory including swap memory. Attention: -1 might cause the system to kill other processes (like the X-Server) to free memory to run the job executable binaries.
- `-k` or `--keep-on-failure`: If a module executable fails to finish execution properly, the temporary files produced by the binary are kept for further analysis. The testbed typically removes all files produced by a failed execution, even if a binary failed for some specific reason, but nevertheless produced valid results.

On a multi processor system one job server for each processor (CPU) can be started to use all processors available.

A job server can be shut down with the usual kill signal (Control-C).

In order to detect crashes or abortion of jobs, the status of each job that has not finished to run after a certain amount of time will be set to 'FAILED'. The duration after which this automatic update happens can be adjusted in the testbed configuration file `TESTBED_ROOT`*/config.php* by changing the following line (time is given in hours:minutes:seconds, see also subsection 3.1.4 on page 69):

```
define('JOBTIMEOUT', '12:00:00');
```

The default value is 12 hours but can be changed arbitrarily. This setting can be made in the user specific configuration file *.testbed.conf.php* located in its home directory, too, overwriting the global settings.

Note that changing the status does not mean that the execution of the process is stopped. Jobs running properly for more than 12 hours will run to an end, only their status will have changed to failed, even if this is not the true status. Nevertheless, the results are contained in the testbed. The only effect of changing the status to 'FAILED' is that the job can be restarted again. In such a case, it is possible that duplicate jobs are running simultaneously. The last one to finish will overwrite all previous results. Hence, the output of the duplicate jobs might be mangled. See the next subsection for how to reset jobs manually.

3.4.5 Maintaining the Database

This subsection discusses some issues related to the maintenance of the testbed databases. The topics are the regular clean up of the database to prevent it from becoming crowded, backup and restoration of the database for security, organizational and communicational reasons.

Database Hygienics

Over time the databases of a testbed installation can become quite big. In order to optimize speed of the database responses and disk space used by a database, deleted entries should be removed from the database persistently eventually. This is not done with a removal command immediately, comparable to the procedure when documents of a file systems are moved to the trash bin upon deletion first. To clean up the database, command `vacuum` can be used:

```
psql -U <username> -c "vacuum" <database>
```

This command garbage-collect and optionally analyze database named `<database>`. If a password is required, the password of the user `<username>` must be entered. User `<username>` needs appropriate permission to do so, though. To find out about the current settings, submenu 'Testbed Status' can be consulted (see subsection 3.3.13 on page 133). For more information about the `vauum` SQL command refer to the documentation of the PostgreSQL database management system [56].

It is advisable to vacuum a database once in a while. On a Unix system the 'cron daemon' can do this automatically. Command `crontab -e` is used to edit a crontab[15]. For example:

```
0 4 0 * * /usr/bin/psql -U postgres -c "vacuum" testbed 1>/dev/null
```

runs the cleanup every Sunday at 4 a.m. For more information about the 'cron daemon' see the man page for `crontab` [71].

The testbed can be reseted by issuing command

```
testbed reset
```

on the CLI. The effect of this command will be that all job with status 'Running' are set to status 'FAILED'. These job can now be restarted which is not able while their status is running, even if they are obviously not running anymore. If the maximum time to run a job has not expired yet and he job finishes execution correctly, its status will be 'FAILED' nevertheless. which would result in a reset of the job, too (compare to the previous subsection). A further discussion can be found in section 4.6 on page 217 about troubleshooting and in subsections 3.3.9 on page 121 and 3.3.8 on page 116. .

[15] A crontab is a list of actions that should be triggered at a special time

Backup and Restoration

For several reasons, it is very useful to be able to backup and subsequently restore the complete contents of a single database of the whole database system at once, not just by exporting single or multiple objects to XML. On the one hand, this provides better security when a database should crash. On the other hand, this can be used to further organize the data by employing different databases for different kinds or projects. Finally, this a very efficient means to communicate experiments as a whole, including specification and results. The complete database currently in use can be exported to a backup with command issued on the CLI:

```
testbed backup
```

This command will use the standard backup mechanism provided by PostgreSQL namely `pg_dump`. It requires to give a user name and possibly a password. The user name typically the one displayed when checking the current testbed status (see subsection 3.3.13 on page 133) which is used to access the database currently in use. Alternatively, the user name and password of the administrator of the database can be given here. The result will be two files containing any data stored by the testbed. These two files, *TESTBED_BIN_ROOT.tgz* and *db.tbz*, will be written to the current directory. The first file contains the executable and module definition files for any module registered as found in directory `TESTBED_BIN_ROOT` (and its subdirectories) in the form of a compressed `tar` file (*.tgz* format). The latter file contains the compressed contents of a dump of the database as resulting from PostgreSQL command `pg_dump` with subsequently running tool `bzip2` to compress the dump. Backup file *db.tbz* can be restored with PostgreSQL command `pg_restore` after having un-zipped it with tool `bunzip2 -k db.tbz` (resulting in file *db.tar*):

```
bunzip2 -k db.tbz
```

```
pg_restore --host=<host> --format=t -d <database> -U <user> -c -O db.tar
```

Host `<host>` is the name of the machine which runs the testbed server installation, `<database>` and `<user>` are the names of the current database and user used. The backup of the binary executables and the module definition files can be restored by extracting the `tar` file with command `tar -xcf TESTBED_BIN_ROOT.tgz` in the root directory, since the file were archived using command `tar -czf` and using absolute path names. Note that extracting archive file `TESTBED_BIN_ROOT.tgz` in the root directory needs proper user rights and might overwrite any existing module binary executables and module definition files. Probably, it is better to extract it in some local directory and then copy the needed subdirectories to the proper location manually. More information about commands `pg_dump` and `pg_restore` can be found on their man pages [77, 78].

This process of restoration of a database backup created by the `backup` mechanism of the CLI tool as described just now has been automated, too. To restore the testbed to a previous backup as produced with command `testbed backup` call:

`testbed restore [<directory>]`

A backup file *db.tbz* (the compressed dump of the testbed database of a previous backup effort using call `testbed backup`) is expected to be in the current directory. The contents of file *TESTBED_BIN_ROOT.tgz* will be restored only if it is contained in the current directory, i.e. it is optional. If asked for it, the appropriate username and passwords of the current user of the testbed (i.e. the user that executes the tool and wishes to restore its currently used database) has to be entered. The command has to be issued with appropriate user rights.

If option `<directory>` is used (`<directory>` denotes a valid path name), then the file containing the module definition files and the binaries (*TESTBED_BIN_ROOT.tgz*) is extracted in directory `<directory>`, otherwise it is extracted to the root directory. Recall that the `testbed backup` command compresses the module definition files and binaries with absolute path names. This might result in loss of data as was described before. Each step of the restoration process will ask for special confirmation on the side of the user.

Finally, more information about how to backup, restore, and maintain the database directly is described in detail in the PostgreSQL documentation, Administrators Guide, Section 8, Backup and Restore [56]. PostgreSQL provides an add-on which enables database administration via a web based graphical user interface called phpPgAdmin. How to install this interface to directly view and perhaps manipulate a testbed database is briefly described in section 4.5 on page 215 and more elaborately in [45].

3.4.6 Display Job Results

By means of the web based user interface of the testbed it is not possible to export any job results individually. They can be incorporated into the exported XML file of an experiment and can be extracted manually form there by copy and paste. However, this is quite cumbersome. To display the output of jobs numbered `<int 1>`, `<int2>`, ..., and `<int n>`, the following command can be used (`<int i>` are integer numbers designating jobs in the testbed, `i` and `n` are natural numbers with `i < n`):

`testbed jobs get <int 1> <int 2> ... <int n>`

The job results are printed one after another separated by three newlines followed by 80 '=' followed by three newlines again to standard output on the console. if a job can not be found, it will be skipped, a warning is printed at the end, though, again separated as if it were a new job result. By using the redirection mechanism `>` or the piping mechanism '|' of the shell, the output results can be stored in a file or further processed.

3.4.7 Display Data Structures

The data structures that are used internally to store the various types of objects of the testbed with the help of PHP language constructs can be printed on demand to standard output, too.

To display the internal data structure of testbed objects call:

`testbed dump <objectidentifier>`

Argument `<objectidentifier>` must identify an object type. Table 3.7 lists all object types and their identifiers to be used with the `dump` command for the individual object types. Compare to subsection 5.2.1 on page 232.

Object Type	<objectidentifier>
Problem Type	`common.soproblemtypes`
Problem Instance	`probleminstances.soprobleminstances`
Module	`common.somodules`
Algorithm	`algorithms.soalgorithms`
Configuration	`configurations.soconfigurations`
Experiment	`experiments.soexperiments`
Job	`jobs.sojobs`
Data Extraction Script	`statistics.soresultscripts`
Analysis Script	`statistics.sorscripts`

Table 3.7: Available object type for displaying their internal data structure

3.5 Organizing and Searching Data

In section 2.2 on page 11, one of the main purpose of the testbed was identified to be the management of the data that accumulate during experimentation. The testbed divided this data into several main types as there are modules, algorithms, configurations, experiments, jobs[16], and data extraction and analysis scripts. The menu structure of the testbed and its tools for searching for data and organizing data were designed accordingly.

Data management does not only include the storage of data. The main question is how to retrieve exactly the data desired. Browsing the data is prohibitive after some time of usage of the testbed, since the amount of data accumulated so far will likely to be too large. As a consequence, powerful tools to search for specific information the user needs are required. Additionally, powerful means to organize the data are necessary.

This section discusses the tools of the testbed that enable the user to search for any type of object contained in the testbed database and that enables the user to organize and manage the data contained in the testbed. Management of data is achieved by enabling a grouping mechanism for objects in the testbed. These groups are called *categories*. This section discusses how to search for objects, i.e. how to generate search filters represented by search queries to the database. Search filter generation is illustrated by some examples. Next, this section explains how to form categories. Category management will be explained and how search filters and categories relate. At the end of this section, some variants of categories, called *global* and *static categories*, are discussed.

3.5.1 Search Filters

Searching for data in a database storing objects means trying to find all objects in the database that are contained in an imagined coherent subset of the set of all objects contained in the database. This sought-after subset is called *target set*. Since digitally stored objects intrinsically can be viewed as consisting of a set of attributes value pairs – name, generation time, links to other objects, contents and so on – objects can only be discriminated by means of their attributes and the corresponding attribute values. This is even more the case if objects are stored in a relational database as is the case of the testbed. Consequently, any coherent subset that might be the desired result of a search finally must be definable in terms of restriction to attributes and the values they are

[16]Note that jobs and their results are related through a one-to-one relation. Therefore, searching for jobs and searching for job results essentially is the same. The only difference is the subsequent usage of either the results or the jobs themselves. Accordingly, the testbed does not distinguish between searching for jobs and searching for job results. Both results and jobs are searched for uniformly under the headline of searching for jobs. If jobs are to be displayed in a submenu by means of categories or the current search filter, actually jobs are retrieved by a job search filter. If job results are required, for example when extracting data, the job results are retrieved.

allowed to adopt, and conditions with respect to how these attribute restriction are to be combined to form further constraints[17]. The objects in the imagined target set shall fulfill these requirements, i.e. restrictions and constraints, in opposition to the objects outside the target set, which do meet the requirements.

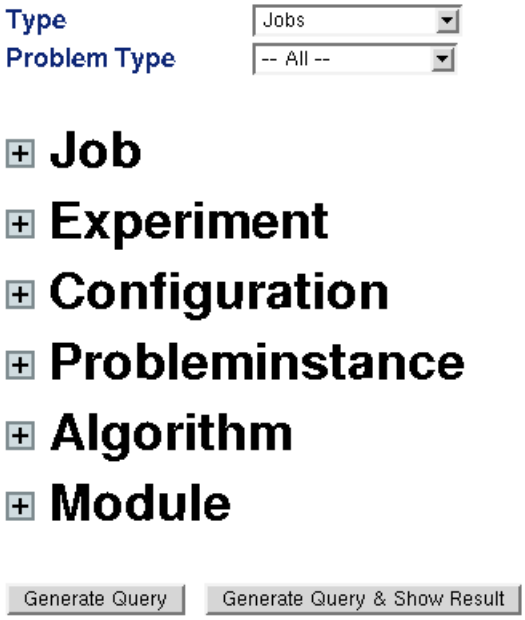


Figure 3.36: Search filter: Generation mask imploded

For example, let the target set be all jobs in the database that have been started yesterday or today and that have finished already. This target set is defined by restricting the values for attribute 'Started' to be 'yesterday' or 'today' and by restricting the values for attribute 'Status' to be 'finished'. These two restrictions are then combined logically AND, that is, only objects that have either value 'yesterday' *or* 'today' for attribute 'Started'*and* that have value 'finished' for attribute 'Status' can be contained in the target set. The process of finding the target set can be viewed as a construction or filtering process. Given the requirements for the target set, each object in the database is checked whether or not it meets the requirements and accordingly whether it is put inside or outside the target set. The set thus constructed by the filtering process need not always be identical with the set imagined, since the requirements actually posed might have been to weak to completely discriminate the target set from the set of all objects. The set resulting from the filtering process is called the *search result* in contrast to the target set. Depending on whether the requirements describing the search result set were correct and complete, the search result and the target set can differ. Note that the search result is the set that was actually constructed by a filter process, while the target set is the intended set.

[17]Of course, this only holds, if the set of all objects is finite, which typically is the case in practice.

The target set can also be viewed as being defined by some kind of prototype for all the objects in the target set. Each object that is consistent with the prototype's attribute values will be in the search result. This imagined abstract is represented by so called *search filters*. Search filter do represent the filtering process. Since the prototype need not be precise enough, a search filter represent the search results rather than the target set. Search filters are imagined in terms of restrictions to attribute values and constraints on combinations thereof. Since in practice objects are stored in a relational database, search filters eventually are implemented or rather represented by SQL statements. These SQL statements implement a search query to the database. The process of filtering becomes that of querying. The search result then will be the query result. Categories as well as the current search filter, as will be discussed later, are stored search filter. The coherence of the target set typically includes that only objects of one type are contained. This is always the case in the testbed. Accordingly, each search filter typically operates only on one type of object, too. This type is called *a search filter's type*. Executing or application of a search filter means executing the representing SQL statement.

Search Filter Generation Tool

In summary, typically the user wishes to find data objects of a given type which fulfill certain conditions. These conditions are expressed in terms of attributes value pairs and constraints on combinations thereof the looked for objects should possess. All objects, which, based on their actual set of supported attributes and the actual attribute values, do not meet the conditions are filtered out. The remaining objects form the set of the search result, called *search result* in short. Within the testbed, the so called *search filters* are used to define conditions on attribute values and necessary relations of attribute values to filter on. Since all objects are stored in a relational database that is accessed via SQL statements, a search filter eventually is expressed and defined in terms of an SQL statement. The testbed provides a tool in the 'Search Filters' submenu that helps to create SQL statements, i.e. queries to the database. To be precise, the tool described here is not directly a search tool, but rather a tool to generate search queries in SQL directly. The advantage of a search query or rather *search filter generation tool* over a direct search tool is that queries generated can be stored and reused later, then working on the current state of the database. These stored search queries are called categories and are explained in subsection 3.5.2 on page 165. Additionally, a direct search tool covering all valid queries of SQL would be too complex for everyday usage. Basically, it would not be easier to use than using SQL directly. Finally, enabling the user to further refine an SQL query after generation allows for construction of an easy to use generation tool that covers the most important and practically most frequently occurring search cases without loosing any power of the SQL query language. The differences between the usage of a direct search tool to a tool for generating search queries are almost

completely transparent if no further refinement of a query generated is needed as in the most cases in practice.

Query by Example

The starting page of the 'Search Filters' submenu is shown in figure 3.36 on page 147.

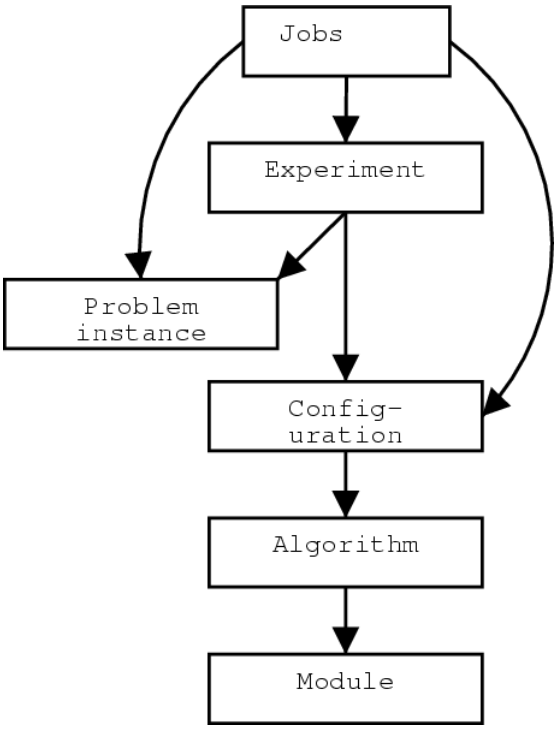


Figure 3.37: Dependencies between data types

The user can specify the search filter to be generated declaratively in a *query by example* fashion [1, 2, 32]. Each attribute of any type of object has been assigned an input field. The user can enter values in these input fields which represents values of the attributes the objects sought-after should possess. The user can enter multiple values at once and can even enter regular expressions in case of text input fields in order to define larger sets of values for attributes[18]. The testbed will then automatically create an SQL statement that combines the restrictions expressed by the value sets defined for the attributes. The values defined for a single attribute will be combined logically OR while the attributes will be related logically AND. That is, an object will be in the search result, if for *all* attributes for which values were specified the object's value of this attribute is in the set of the values specified. For example, if all jobs are sought which were generated either at time 'A' or 'B' and which ended either at time 'C' or 'D', specifying this query can

[18]The usage of regular expressions for filling out text input fields is explained later in this subsection on page 162.

be done by entering values 'A' and 'B' into the input field for attribute 'Generated' and values 'C' and 'D' into the input field for attribute 'Ended'. This approach to querying is called *query by example* because by specifying required values for some attributes a virtual prototype or example is specified for the objects of the target set.

Search Filter Generation

The search filter generation mask is designed according to the data types of the testbed as can be seen in figures 3.38 on the facing page and 3.39 on page 152. All attributes existing for any kind of objects are grouped together. By clicking on the + sign in front of the headlines representing the various kind of objects, a more detailed section, each with several input fields for featured attributes can be expanded[19]. An expanded section can be imploded by clicking on the appearing - sign. Typically, only objects of the same type are wanted as well as typically only objects related to the same problem type are wanted. For this reason, the attributes for type and related problem type – which almost all objects possess – are factored out and put at the beginning of the page.

When creating a search filter by means of the 'Search Filter' submenu, the object type to operate on is selected with selection box labeled 1 in figure 3.36 on page 147. Next, the user can specify which problem type the search results should be related to with selection box labeled 2 in the same figure. Afterwards, further conditions on single attribute in the form of attribute values can be specified by entering them in the appropriate input fields. After the user has entered all values in the various input fields[20], the SQL statement implementing the search filter will be generated by pressing button 'Generate Query' (see figure 3.40 on page 153). The query generated from the user input is shown in a new box at the end of the page. The user can edit the query in this box. For example, since the attribute value restrictions are linked logically AND by default, the user can replace AND with OR statements in the query or set parenthesis around conditions. Also, conditions can be duplicated and joined logically AND or logically OR. That way arbitrary logical formulas can be formed. By pressing button 'View Result' the possibly refined query is performed on the database and the result is shown on a separate page to be reviewed. The upcoming page presents the search result and is organized in the same way as the submenu for the search result object type is organized. To be precise, it basically is the submenu, simply with the search filter generated applied as so called current search filter as category filter. When executing an SQL statement generated, the represented search filter automatically is stored as new current search filter. By pressing the browser 'Back' button the user can go back to the search filters page and

[19]This is only possible if the Browser is DOM (Document Object Model) capable. If the browser is not capable of DOM, all sections will be displayed expanded by default.

[20]The attribute names allude to the different attributes that were presented ex- or implicitly, for example by column names, when discussing the various submenus for the various types of objects in section 3.3 on page 95

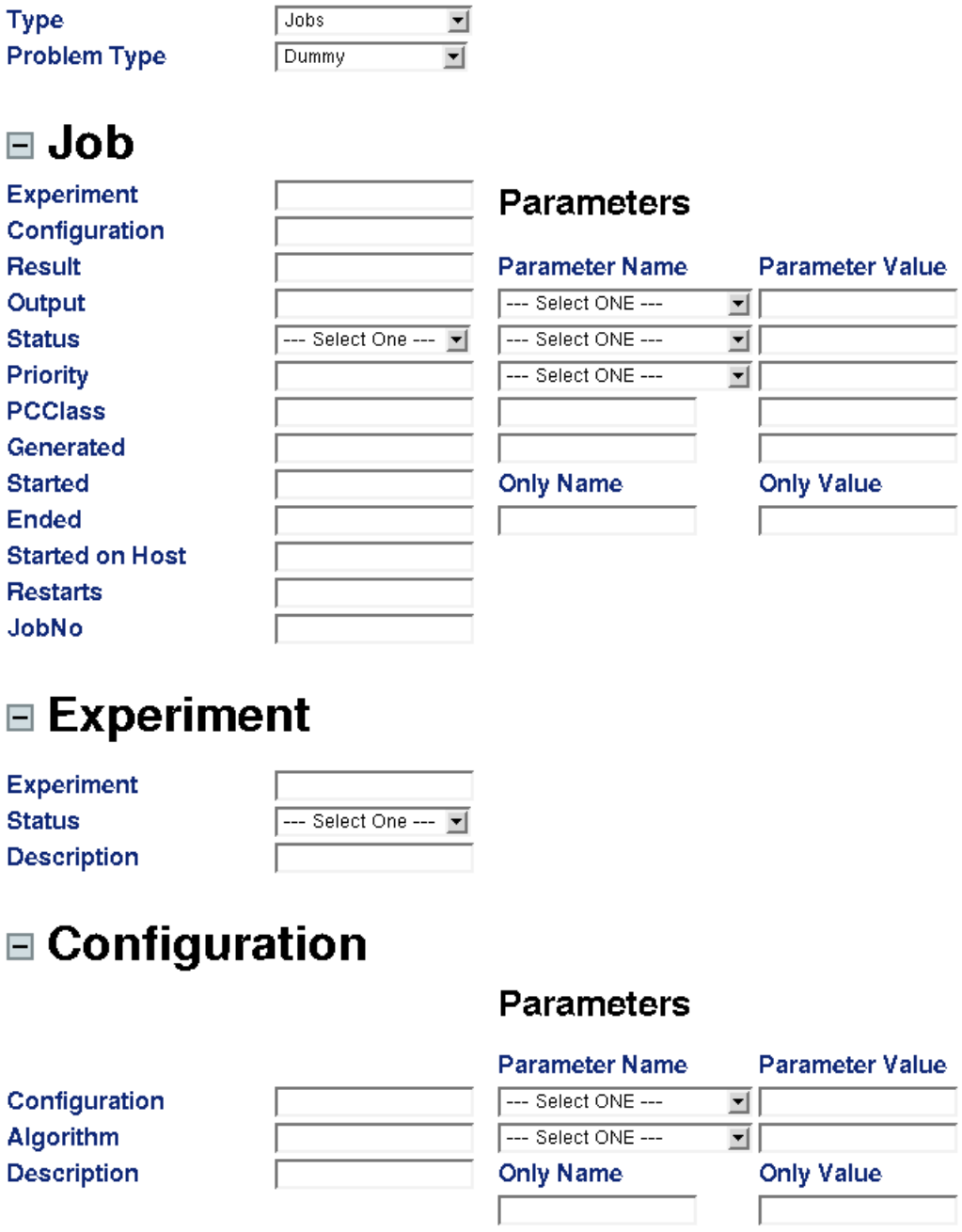


Figure 3.38: Search filter: Generation mask expanded – top

modify the search filter, either by changing values in the input fields and a subsequent new query generation or by editing the SQL statement in the text box containing. This processes of generation, editing and testing can be repeated arbitrary times. The process of first generating the SQL statement of a query and then submitting it can be shortcut by pressing button 'Generate Query & Show Results'. If the query generated is not to be changed anyway, pressing button 'Generate Query & Show Results' generates and

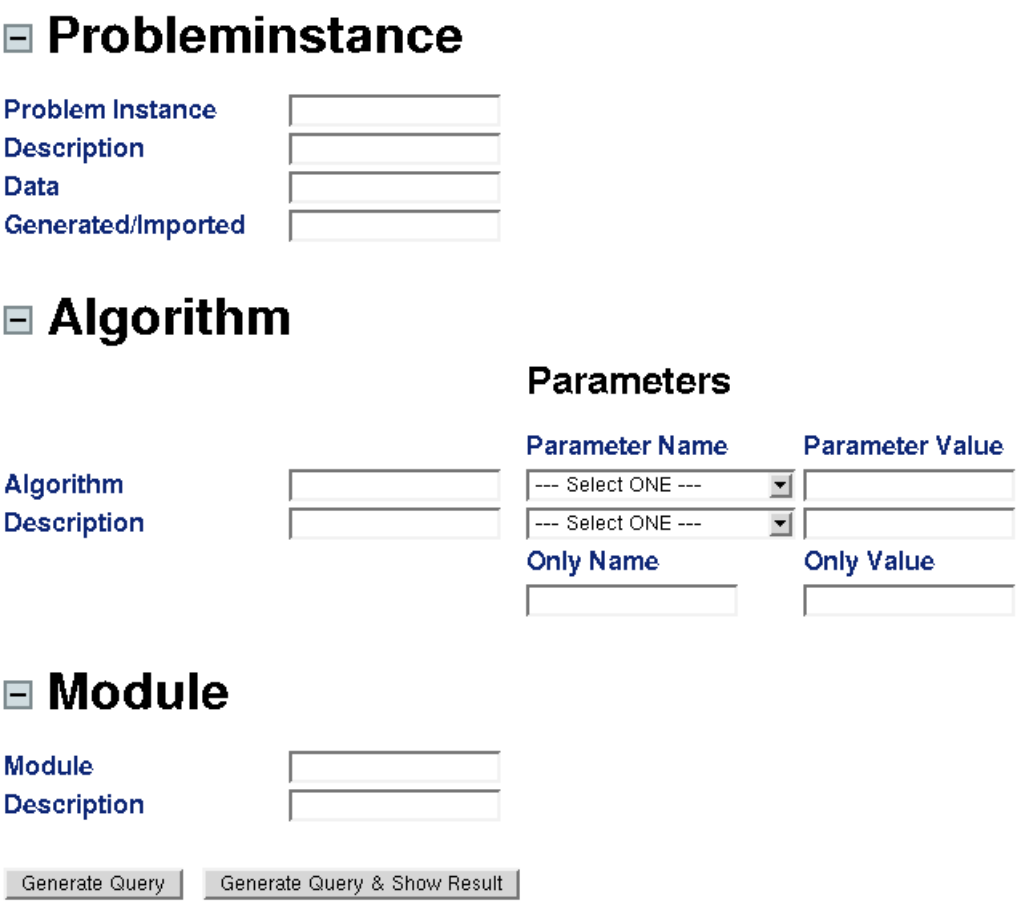


Figure 3.39: Search filter: Generation mask expanded – bottom

executes the query at once (see figure 3.40 on the next page).

If the user finally is content with the search filter, it can be saved again as either the so called current search filter or as a category. Both types of filters are stored search filters and can be used in submenus and when extracting data (see part one of subsection 3.3.2 and 3.3.10 on pages 96 and 3.3.10, respectively). The difference is that a category is stored permanently by the testbed and has to be removed manually, while the current search filter, as the name indicates, will be overwritten any time a new current search filter is saved or a newly generated search filter query is executed. A search filter is saved as current search filter by pressing button 'Save Filter'. A search filter is saved as new category by pressing 'Save as Category' after entering a name for the new category in the input field next to this button (see figure 3.40 on the facing page). A separate current search filter for each kind of objects that can be retrieved with search filters will be stored and applied appropriately. The current search filters for the different object types can be viewed in submenu 'Show' of submenu 'Search Filters'.

As will be explained soon when discussing categories, categories can be organized hierarchically. Hence, it is also possible to arrange a new category directly in the category

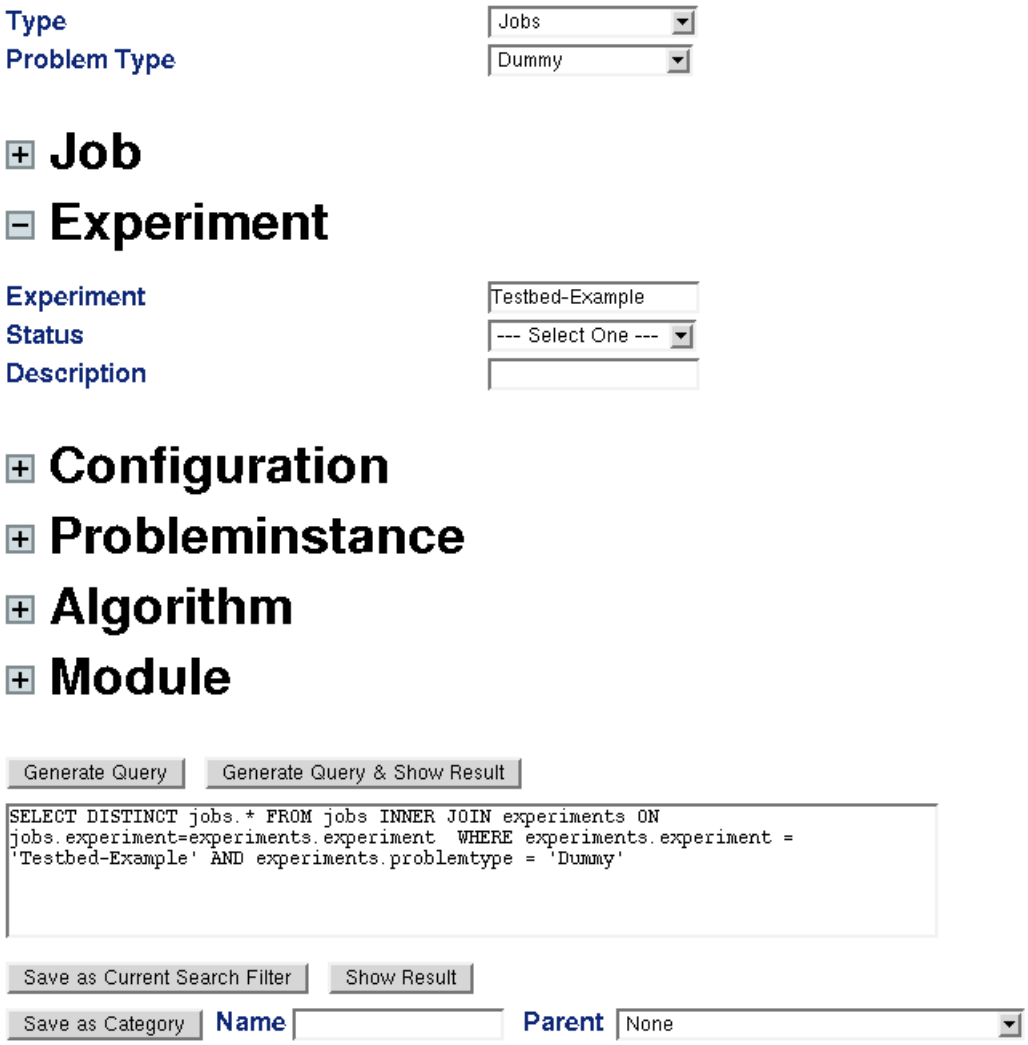


Figure 3.40: Search queries

hierarchy by selecting a parent category. This can be done by selecting the desired parent category from the proposals of the selection box just right to the text input field for entering the name of the new category. Categories created such are dynamic[21].

Derived Attributes

Object of various types are related and dependent on each other in many cases and ways. Figure 3.37 on page 149[22] illustrates the dependencies between the different types of objects. A relation between two types of objects exist, if either one object type, possibly

[21] This is in contrast to static category which exist, too, and which will be explained later in the subsection.

[22] It is nearly the same as figure 2.1 on page 7, only upside down. Since scripts of any kind are not related to any other kind of object they are omitted.

transitively, depends on the other or if they are related to the same type. Based on this definition, all object types displayed in figure 3.37 on page 149 are related to each other. The notion of relations between object types allows for a broader definition of attributes. By means of relations, so called *derived attributes* can be introduced. If one type of object is related to another type of object, the attributes of the related type become derived attributes of the first type. These derived attributes can also be used to specify queries on a query by example basis. Therefore, all input fields of the search mask can be used to specify search filter for any type of object wanted. Attributes belonging to an object type directly are called *direct attributes* and are grouped together in the search mask in the section headlined by the type.

Some examples will illustrate the need of derived attribute in order to discriminate some very frequent and basic target sets.

Example 1

Consider the following target sets:

1. All jobs that are based on algorithm `A`.
2. All jobs that have been run on problem instance `B`.
3. All jobs whose algorithm contains a module named `C`.
4. All jobs whose algorithm has run with a parameter with name `D` set to any value.
5. All configuration that are based on algorithm `E`.
6. All problem instances that are used in experiment `F`.
7. All algorithms that were run an problem instance `G` in some experiment.

When searching for objects of a specific type, not only direct attributes for objects of this type can be used, but derived attributes as well. The query generated will, via the SQL 'JOIN' operator, relate derived attributes with direct attributes and generate an appropriate query. Again, the attribute value restrictions are connected logically AND while the values of each attribute are connected logically OR. Deviation from this standard have to be dealt with by manipulating the query generated directly. For example, querying for the set of all jobs that have been run on problem instance 'H' *or* that are based on algorithm 'I' can not be generated directly. Instead, the AND connection in the SQL statement relating the conditions on attributes name for both object type problem instance and algorithm has to be changed in an OR connection.

The example target set from example 1 can be specified as search filter as follows. Only the attributes used and their required values are given. An arrow '->' separates attribute or field names from type or headline an attribute belongs to.

1. Select = Jobs
   Algorithm -> Algorithm = `A`
2. Select = Jobs
   Problem Instance -> Problem Instance = `B`
3. Select = Jobs
   Module -> Module = `C`
4. Select = Jobs
   Job -> Parameter Name = `D` (via selection box, field Parameter Value after and left empty) or Job -> Only Name = `D`
5. Select = Configurations
   Algorithm -> Algorithm = `E` or
   Configuration -> Algorithm = `E`
6. Select = Experiments
   Problem Instance -> Problem Instance = `F`
7. Select = Algorithms
   Problem Instance -> Problem Instance = `G`

Parameter Handling

One major complication when generating search filters is how to deal with parameters and their settings. For example, wanting a target set that is based on parameters of modules, only the parameter name can be used as a discrimination criterion and not any values, since module parameters have no value assigned. On the other hand, with respect to parameters of configurations their names and values are of interest. Algorithms, in turn, lie somewhere in between configurations and modules, since they can have default values set for parameters, but only one value can be defined per parameter, while configurations can have sets of values defined for any parameter. In order to handle these differences properly, the various input fields for parameter name and value constraints for the miscellaneous types of objects will have different meanings. The rules are as follows:

1. Modules do not have input fields related to parameters, since module parameters are only of interest in connection with algorithms and can be handled sufficiently there.
2. Potentially, one parameter input field for each parameter and its value setting(s) is available, even if the search filter generation mask does only provide a limited number of such input fields. This is because the number of these input fields can be changed (see subsection 5.3.2 on page 270).

3. The set of all input fields for parameters is divided into several groups. One group provides a selection box to the user where the user can choose a name from among all parameters known to the testbed related to the problem type selected. Attached to this selection box is a text input field to enter values for the parameter. These input fields are named 'Parameter Name' and 'Parameter Value', respectively. Input fields 'Parameter Name' and 'Parameter Value' only work properly if used together. Another group of parameter input fields enables the use to enter both parameter name and value as strings, possibly using regular expressions for both. This kind of parameter input field is only available referring to jobs and is located beneath the input field of the first group of parameter input fields. The last group of input fields provide for entering a name or a value for a parameter only, requiring any object in the search result to have set a parameter with the entered name or requiring a parameter with arbitrary name who has been assigned the value entered, respectively[23]. These parameter input fields are labeled 'Only name' and 'Only Value', respectively.

4. Depending on the object type parameter input fields are assigned to, they work on different repositories of parameters:

   - Parameter input fields assigned to type algorithm work on basis of hidden parameters of algorithms. That is, the selection boxes will only propose hidden parameters. Any value restriction entered thus can only refer to hidden parameters.
   - Parameter input fields assigned to configurations and jobs work on all parameters that have be configured, i.e. provided with a value. Parameters that have not been configured will not be used by the testbed when executing a job and consequently will not be considered for the restriction induced by the values entered in any parameter input field for types job and configuration.
   - Parameters which have been set to a default value can not be configured in a configuration and consequently will not be considered in the repository of parameter used for type configuration. Nevertheless, they show up as parameters in connection with jobs, since these parameters are considered to be set by the testbed. Therefore, they will be in the repository of parameters used for jobs only.

Note that the parameter names refer to the names as viewed by the testbed. Recall from subsection 3.3.6 on page 107 on page 110 that parameter names as viewed by the testbed are constructed from the parameter name as exported by the module definition files, the position in the sequence of modules of an algorithm and the algorithm name. If the user wishes only to refer to parameter names as exported by the module definition files, wildcards and regular expressions as explained later can be used.

[23] Value sets can be applied here, too.

Example 2

In order to exemplify how to use parameter attributes the following examples are presented (notation is as in example 1)[24]:

- All jobs that have run with a parameter named `I`:
  Select = Jobs
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `*` or
  Job -> Parameters -> Only Name = `I`

- All jobs that have run with parameters named `I` and `J` set:
  Select = Jobs
  Job -> Parameters -> Parameter Name = `J` and Parameter Value = `*`
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `*`

- All experiments that have created a job that has run with parameters named `I` and `J` set:
  Select = Experiments
  Job -> Parameters -> Parameter Name = `J` and Parameter Value = `*`
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `*`

- All jobs that have run with parameters named `I` and `J` set with values `II` and `JJ`, respectively:
  Select = Jobs
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `II`
  Job -> Parameters -> Parameter Name = `J` and Parameter Value = `JJ`

- All algorithms that have run on any problem instance with parameters named `I` and `J` set with values `II` and `JJ`, respectively:
  Select = Algorithms
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `II`
  Job -> Parameters -> Parameter Name = `J` and Parameter Value = `JJ`

- All algorithms that have run on problem instance `K` with parameters named `I` and `J` set with values `II` and `JJ`, respectively:
  Select = Algorithms
  Job -> Parameters -> Parameter Name = `I` and Parameter Value = `II`
  Job -> Parameters -> Parameter Name = `J` and Parameter Value = `JJ`
  Problem Instance -> Problem Instance = `K`

- All problem instance on which an algorithm has been run in any experiment with a parameter named `L`:

[24]Wildcard '*' is a regular expression construct representing an arbitrary substring. It is discussed in the next subsection,

Select = Problem Instances
Job -> Parameters -> Only Name = `L`

- All problem instance on which an algorithm with any parameters set to value `L` has run in any experiment:
  Select = Problem Instances
  Job -> Parameters -> Only Value = `L`

- All configurations that configured a parameter named `M`:
  Select = Configurations
  Configuration -> Parameters -> Only Name = `M`

- All configurations that configured a parameter named `N` with value `NN`:
  Select = Configurations
  Configurations -> Parameters -> Parameter Name = `N` and Parameter Value = `NN`

- All experiments that use a configuration that has configured a parameter named `N` with value `NN` in algorithm `O`:
  Select = Experiments
  Configuration -> Parameters -> Parameter Name = `N` and Parameter Value = `NN`
  Algorithm -> Algorithm = `O`

- All problem instances that were processed by an algorithm with any parameter set with value `P`:
  Select = Problem Instances
  Job -> Parameters -> Only Value = `P`

- All algorithms with a hidden parameter named `Q`:
  Select = Algorithms
  Algorithm -> Parameters -> Only Name = `Q`

- All experiments that use an algorithms with a hidden parameter named `R` which was set to value `RR`:
  Select = Experiments
  Algorithm -> Parameters -> Parameter Name = `R` and Parameter Value = `RR`

- All jobs that are based on an algorithm who has any hidden parameter set to value `S`:
  Select = Jobs
  Algorithm -> Parameters -> Only Value = `S`

As can be seen from some examples, the 'Only Name' and Only Value' fields are not really needed. Filling field 'Only Name' with `xyz` is equivalent to filling a field 'Parameter

Name' with `xyz` and its corresponding field 'Parameter Value' with `*`. Filling field 'Only Value' with `xyz` is equivalent to filling a field 'Parameter Value' with `xyz` and its corresponding field 'Parameter Name' with `*`.

Search filter refinement

When using the search filter generation tool exclusively, the user does not need to know anything about the internal database structure of the testbed. The references between the different types of objects in form of joins of database tables storing the object type are resolved automatically by the testbed. As there are too many possibilities how to combine individual attribute value restrictions only the above mentioned logical AND conjuction is made by the testbed.

If this type of connection is not sufficient, for example if all algorithms are wanted that have name `A` and have been used in experiment `C` or that have name `B` and have been used in experiment named `D`, the user can still use an automatically created search filter as a starting point for refinement of its SQL query in cases a more complex query is needed. In almost no cases, the user has to change anything with respect to the joins contained in the SQL statement used as starting point. In short, everything before the `WHERE` construct of an SQL statement generated should not be changed. In order to generate an appropriate starting point for a search filter refinement, the user must ensure that all object types that have an attribute contributing to the target set specification are joined in the SQL statement. This can easily be achieved by filling at least one attribute input field for each such object type contributing. This will include proper joins, the rest can be changed arbitrarily then.

In case of the example presented just now, the following first search filter specification will yield the SQL query listed beneath the specification:

Select = Algorithms
Algorithm -> Algorithm = `A`
Experiment -> Experiment = `C`

```
SELECT DISTINCT algorithms.* FROM algorithms
    INNER JOIN configurations ON algorithms.algorithm=configurations.algorithm
    INNER JOIN expusesconf ON configurations.configuration=expusesconf.configuration
    INNER JOIN experiments ON expusesconf.experiment=experiments.experiment
WHERE experiments.experiment = 'C' AND algorithms.algorithm = 'A'
```

Changing the `WHERE` part of this SQL statement as follows will lead to the correct SQL statement:

```
WHERE (experiments.experiment = 'C' AND algorithms.algorithm = 'A')
    OR (experiments.experiment = 'D' AND algorithms.algorithm = 'B')
```

Note that any input fields other than selection boxes are text input fields. Thus, only strings can be entered. For attributes featuring numerical values, the number will be considered as strings. Arithmetical comparisons are not possible that way. In order to remedy this for a specific search filter, manual query refinement has to be performed. For example, if all jobs that with job number between `1` and `20` are wanted, the following search filter specification can be used to generate an SQL query that can then be further refined to yield the proper query:

Select = Jobs
Job-> Job No = `1`

SQL statement generated:

```
SELECT DISTINCT jobs.* FROM jobs
    INNER JOIN experiments ON jobs.experiment=experiments.experiment
WHERE jobs.job = '1'
```

Refined `WHERE` part:

```
WHERE jobs.job >= 1 AND jobs.job <= 20
```

Of course, as will be seen later when discussing the usage of regular expressions, the search filter could have been specified easily as follows

Select = Jobs
Job-> Job No = `1 ... 20`

resulting in the generation of the following SQL statement:

```
SELECT DISTINCT jobs.* FROM jobs
    INNER JOIN experiments ON jobs.experiment=experiments.experiment
WHERE jobs.job BETWEEN '1 ' and '20'
```

Time intervals are specified similar. For example, if all jobs that were started between the first of January 2003 and the first of February 2003 are demanded the following search filter specification can be used to generate an SQL query that can then be further refined to yield the proper query:

Select = Jobs
Job-> Started = `1`

SQL statement generated:

```
SELECT DISTINCT jobs.* FROM jobs
    INNER JOIN experiments ON jobs.experiment=experiments.experiment
WHERE jobs.started = '1'
```

Refined `WHERE` part:

```
WHERE jobs.generated >= '2003-01-01' AND jobs.generated <= '2003-02-01'
```

Timestamps can be viewed as strings in a special format. For mor information about the timestamps format in PostgreSQL see [56]

Instead of changing the number of parameter input fields permanently by changing the templates for the search mask as described in subsection 5.3.2 on page 270, having too little parameter input fields available for a special search filter request can be remedied by search filter refinement, too. This is done by simply copying and next adding the comparisons after the `WHERE` construct in the SQL statement which represent the attribute value restrictions for the group of parameter input fields that were not numerous enough. For example, if all jobs are wanted that have at least three parameters set. The name of the first parameter should contain substring 'yMax', the name of the second should contain substring 'yMin' , and the name of the third parameter should containing substring 'randomY'. The refinement process could look like the following (Wildcard '*' is a regular expression construct representing an arbitrary substring. It is discussed in the next subsection):

Search Filter:

Select = Jobs
Job -> Parameters -> Parameter Name = `Dummy_1_yMax` and Parameter Value = `*`
Job -> Parameters -> Parameter Name = `Dummy_1_yMin` and Parameter Value = `*`
Job -> Parameters -> Parameter Name = `Dummy_1_randomY` and Parameter Value = `*`

Generating the SQL statement for this query will yield:

```
SELECT DISTINCT jobs.* FROM jobs
    INNER JOIN experiments ON jobs.experiment=experiments.experiment
    JOIN jobparameter jobparameter1 ON jobparameter1.job=jobs.job
    JOIN jobparameter jobparameter2 ON jobparameter2.job=jobs.job
WHERE (jobparameter1.parametername = 'Dummy_1_yMax' AND jobparameter1.value ~~* '%')
    AND
    (jobparameter2.parametername = 'Dummy_1_yMin' AND jobparameter2.value ~~* '%')
```

Now, this can be refined to:

```
SELECT DISTINCT jobs.* FROM jobs
    INNER JOIN experiments ON jobs.experiment=experiments.experiment
    JOIN jobparameter jobparameter1 ON jobparameter1.job=jobs.job
    JOIN jobparameter jobparameter2 ON jobparameter2.job=jobs.job
    JOIN jobparameter jobparameter3 ON jobparameter3.job=jobs.job
WHERE jobparameter1.parametername ~~* '%yMax%' AND
    jobparameter2.parametername ~~* '%yMin%' AND
    jobparameter3.parametername ~~* '%randomY%'
```

If the SQL statement refined is erroneous, the upcoming page when submitting the query will be the search mask instead of an overview over the search results. If the search result is empty, no entries will be displayed in the overview page.

For further information see 5.1 on page 227 for a detailed description of the database structure of the testbed and see subsection for 5.1.1 on page 228 how to create queries

directly from the database structure as shown in figure 5.1 on page 229. Resources concerning the SQL query language are [35, 26, 56], while databases are described in [36, 34, 33, 32, 31, 30].

Note that all SQL queries are run unchecked on the database. If the user uses a DELETE statement in the query (no DELETE statement is created automatically, though) all data meeting the conditions and the data that depend on these database entries will be deleted!

Wildcards and Regular Expressions

All text input fields of the search mask and the text input field of the regular expression filter in submenus (see subsection 3.3.2 on page 96 and item 5.2 in figure 3.17 on page 100 support the use of regular expressions and wildcards. That way, sets of attribute values can be defined easily. Typically, using regular expressions or wildcards when filling a text input field defines a set of strings. The single elements of this set are logical OR connected. That is, given a specific attribute, any object whose value for this attribute matches the regular expression, i.e. is in the set of possible values defined by the regular expression, fulfills the restrictions for this attribute as defined by the regular expression. Of course, for an object to appear in the search result for a search filter, it typically must meet *all* other attribute restrictions as well as described previously. This paragraph describes and discusses the use of regular expressions and wildcards in text input fields. Again, this is relevant for any text input field featuring regular expression.

The testbed accepts different types of wildcards and regular expressions. These are described next in the form of a list together with their associativity. Note that the different types of wildcards and regular expressions can not be mixed and may lead to an unexpected result. Some examples will clarify how to use wildcards and regular expressions. In dependence on the appearance of special characters in a regular expression, it is assumed to be of one of the possible types listed next automatically. The rules for this automatic type, detection are given after a short explanation of the regular expression's syntax and semantic.

- POSIX regular expression

  The first type of regular expressions supported by the testbed are standard POSIX regular expressions and ANSI-SQL LIKE patterns. The syntax of a POSIX regular expression is `operator '<regular expression>'`. The regular expression itself must be enclosed between the two ''(apostrophe) Operators can be:

  = Matches only the exact string value that is given as argument. No special characters are known except for the backslash that can be used to escape a '' or the backslash itself.

~ Matches using regular expressions, i.e. treats the argument as a regular expression and not as a literal string (as operator '=' does). Patterns are matched over any substring in contrast to the mode of operation of ANSI-SQL LIKE patterns. This operator is set by default when a regular expression entered in a text input field is recognized as a POSIX regular expression (see later).

~ ~ Matches using ANSI-SQL LIKE patterns. These patterns are provided by the SQL language. Patterns in ANSI-SQL LIKE can contain only two different wildcards, namely '_' matching a single character and '%' matching for zero or more single characters. ANSI-SQL LIKE patterns matches always cover the entire string. In order to match a pattern within a string, the pattern must start and end with a '%'.

! as a prefix of an operator inverts the search, i.e. all elements not matched by the regular expression are returned.

***** as a suffix of an operator makes the search case- insensitive. This is set by default when a regular expression entered in a text input field is recognized as a POSIX regular expression (see later).

Altogether, the following combinations of operators, pre- and suffixes are possible: '=', '~', '~*', '~~', '~~*', '!=', '!~', '!~*', '!~~', and '!~~*'. The syntax and semantic of POSIX regular expressions is described in short next. Note that this is only a coarse overview. See the man page [74] `man 7 regex` for a detailed description of usable regular expression. More information about POSIX regular expressions can also be found in [6], [67] and [75].

- A single character, an escaped special character, any regular expression in brackets '(' and ')' or a list of characters in brackets '[' and ']' is called an atom. A single non-special character matches this character, a single escaped special character matches this special character, a regular expression in round brackets matches what the regular expression without round brackets would match.
- An atom can be followed by a '*', a '+', a '?' or an expression `{a,b}`. The first matches zero or more matches of the preceeding atom, the second matches one or more matches of the preceeding atom, the third matches zero or one matches of the preceeding atom, while the last matches between `a` and `b` matches of the atoms, provided `a` and `b` are integers greater or equal to zero with $a \leq b$.
- '(' and ')' can be used to bracket parts of the regular expression to use as atom.
- '|' can be used to enumerate various options.

  - Brackets '[' and ']' enclose a list of characters indicating to match any of these. If the list begins with a '^', it matches any character not from the list. Two characters separated by '-' is a shorthand for the full range of characters between them according to the ASCII enumeration. To include a ']' or a '-' in the list, it must appear as first character.
  - Within a list of characters, the name of a character class enclosed in '[:' and ':]' represents all characters belonging to the class. Standard character class names are for example: `alnum`, `digit`, `punct`, `alpha`, `graph`, `space`, `blank`, `lower`, `upper`, `cntrl`, `print`, and `xdigit`. These reflect the character classes as defined in [75]. A character class can not be used as a bound of a range (as defined by a '-').
  - A backslash '\' escapes any special character. These are '^', '.', '[', ']', '$', '(', ')', '|', '*', '+', '?', '{', '}', and '\'.
  - Special characters '^' and '$' match the begin and the end of a line, respectively.

- Shell pattern

  Patterns as used in a Unix shell can also be used for searching. All shell pattern searches are case-insensitive. A symbol '?' represents any single character, '*' matches zero or more characters. Shell patterns matches whole strings instead of arbitrary substrings. Shell patterns have been added because they are more intuitive than the POSIX regular expression. However, they are less powerful than the POSIX regular expressions. Internally,shell patterns are converted to a POSIX regular expression.

- Range expression

  To specify a range, e.g. the range between number `a` and `b`, the user must enter `a ... b` into the text-field. It is searched for all dates or strings which are between `a` and `b`. A range is a shortcut for `>= a` AND `<= b`. The range delimiters can be arbitrary integer or real numbers or they can be arbitrary strings. In the first case, the usual arithmetic order is given, in the latter case lexicographical order is given.

- Comparison

  The usual comparison for strings and numbers like `<`, `<=`, `<>`, `>=`, `>` are also supported. The order is as for the previous item.

- Fixed string

  If none of the above search patterns applies, an exact search for the entered string will be attempted.

Automatic detection of regular expression type works as described in what follows. Several rules exist to assign a regular expression type to a given user input which are evaluated in turn. The rules are checked in the same order as listed next. As soon as a rule fits, the according regular expression type is set and the whole regular expression is evaluated with this type in mind, possibly yielding incorrect or misleading results or non at all, if the regular expression syntax of the type chosen is violated.

- If user input begins with '=', '~', '~*', '~~', '~~*', '!=', '!~', '!~*', '!~~', or '!~~*', it is treated as a POSIX regular expression. The expression is set to be case-insensitive by default, i.e. operator '~*' is used implicitly.
- If a user input contains any of the characters '^', '|', '$', '[', or ']', it is taken to be a POSIX regular expression, too. The default is case-insensitive, too.
- If a user input contains characters '?' or '*', it is assumed to be a shell pattern. These are translated automatically into ANSI_SQL LIKE patterns with '_' replacing '?', and '%' replacing '*'. The resulting expression is case-insensitive.
- Range expressions must contain a range expression '`floor... roof`' which is automatically converted into an SQL expression 'BETWEEN `floor` AND `roof`'.
- A comparison must begin with a comparison operator '<', '<=', '<>', '>=', or '>'. Matching again is case-insensitive, strings are ordered according to the conventional lexicographic ordering.
- Finally, if no previous rule matches, the user input is taken as a fixed string, matching case-insensitive again.

The input of the user is interpreted in the order as listed above. The first type of regular expression recognized will be used. Therefore, the different types of wildcards can not be mixed. Be aware that mixed regular expressions might still be interpretable as one of the forms discussed just now. However, it is likely that the results for the expression entered is not what was intended by the user. In particular, be aware of the placement of white spaces in regular expression. This can falsify a regular expression enormously.

Example 3

According to the description given before, table 3.8 on the following page lists some examples that show how to use wildcards and regular expressions.

3.5.2 Categories

As was mentioned briefly before, *categories* essentially are stored search filter. Given an encoding of the requirements the objects in the target set should comply with, i.e. given

Input	Effect
`~* 'foo'`	Matches foo, Foo, FOO, ..
`~* 'b+'`	Matches all words containing an `b`.
`(foo|bar)`	Matches the word foo or bar case-insensitive.
`~ '(foo|bar)'`	Matches exactly the words `foo` or `bar` nothing more, nothing less.
`foo*`	Matches any string starting with foo, Foo, ..
`20 ... 30`	Matches all values between 20 and 30.
`Aachen ... Munich`	Matches all strings between Aachen and Munich (e.g a 'A' will not match whereas a 'Mannheim' will match).
`< 20`	Matches all values less than 20.
`foo`	Matches exactly the word `foo`.
`*foo%`	This expression mixes ANSI-SQL-LIKE constructs and Shell-patterns. It may not work. The intended meaning is to match text starting with `*foo`.

Table 3.8: Wildcard examples

a search filter definition, the filter process can be repeated arbitrarily if its specification is stored. Stored search filter are called categories. If the filter process is performed on a regular basis, e.g. when a search filter is applied to present submenu entries to the user, the resulting set will always be based on the last state, i.e. contents of the database. In this respect, categories provide means for dynamically build subsets of objects in the database and hence provide means to organize the data contained in the database flexibly and dynamically. This technique has been denoted a view on the database in the domain of relational databases. It has also be used in the Presto system [38] where it was called fluid collections. Instead of organizing data statically in hierarchical structures as is done with the file system of modern operating systems, categories provide a far more flexible mean to organize data.

Categories come in several forms. Typically, they are dynamic as explained just now by defining an SQL statement. If this is the case, a category is called *dynamic category*. On the other hand, if no defining SQL statement is provided, categories are called *static categories*. Objects can be assigned explicitely and statically by the user to static categories that way grouping sets of data without having to form a potentially complicated SQL statement. Adding elements statically to a dynamic category is not possible. Additionally, categories can be divided into *local categories* that are only usable for a specific object type and *global categories* that can contain arbitrary objects. Global categories

can be applied wherever categories can be applied. When using global categories, only those members belonging to the application context, e.g. in submenus, will be displayed. Finally, categories can form a hierarchy. This can be done by entering another category as parent whenever a category is created.

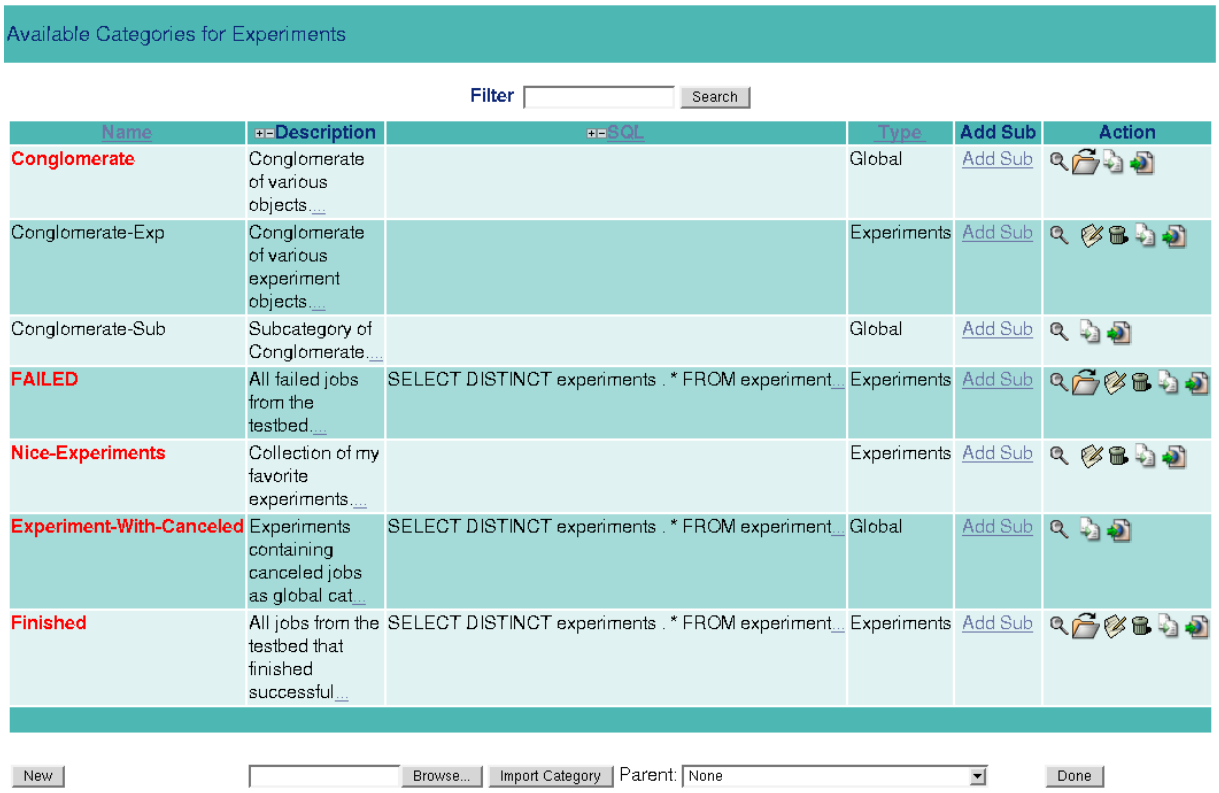


Figure 3.41: Categories submenu for experiments

Any local categories for the different types of objects can be viewed in the 'Categories' submenus of the corresponding submenu (see figure 3.41 for an example for the 'Experiment' submenu). All global categories are viewed in submenu 'Global categories' and will show up in any object type specific category submenu, too. All category submenus feature columns named 'Name', 'Description', 'SQL', 'Problem Type', 'Add Sub', and 'Actions'. The entries for column 'Type' indicate the object type the categories operates on. In case of a global category, which can show up in the local categories submenus, too, this is 'Global'. Column 'SQL' displays any SQL statement that might implement an entry, i.e. a category. Categories can be edited, deleted, and exported to XML and consequently imported from XML, too. Global categories can only be edited or deleted in submenu 'Global Categories'. Local categories can be set to be the current search filter as will be discussed soon. Categories which are based on a parent category are not highlighted, while categories with no parent are displayed in red. The import procedure works as in the other submenu only this time a parent category can be specified the imported category is assigned to. A parent category is assigned when importing a

category by selecting the desired parent category in selection box named 'Parent'. The detailed view of a category presents any parent, the object type (or 'Global' in case of a global category), the name, a description and a possible SQL statement (see figure 3.42 on page 168). The same view is presented to the user when an existing category is to be edited. However, only the description or the SQL statement can be changed. Changes are submitted by pressing button 'Change'. It is not possible to view the elements a category represents in the detailed view. In order to do this, one simply has to use the category as filter in the according submenu.

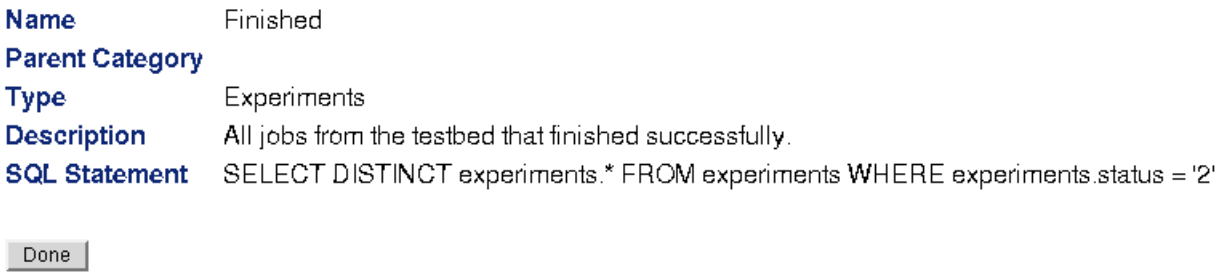


Figure 3.42: Detailed view of a category

New categories can be added from scratch in the 'Categories' submenus, too, by pressing button 'New'. On the upcoming page (see figure 3.43 on the facing page), the user can select a parent category for the new category in selection box labeled 'Parent Category' (or none). A name and a description are entered in text input fields 'Name' and 'Description', while in text input field 'SQL Statement' the user can enter an SQL statement that is to implement the category. If no SQL statement is used here, the category will become static, otherwise it will be a dynamic category. Note that the SQL statement entered is not validated. Improper statements will yield errors later when the category is applied. A new category finally can be stored in the testbed by pressing button 'Save Category'. The page can be reset by pressing button 'Reset'. Button 'Cancel' leads back to the 'Categories' submenu.

Dynamic categories can be used to set the current search filter. In this case, the SQL statement implementing a category is copied to implement the current search filter. This can be done by clicking on the appropriate action in a 'Categories' submenu (see table 3.2 on page 99, icon) or in submenu 'Set From Categories' in submenu 'Search Filters' (see figure 3.44 on page 169).In the latter case, selection box 'Set Filter for' provides a list of all object types that have at least one local category defined. The user can chose the type of object for the current search filter to set from. The second selection box 'Category' then presents type dependent a list of all available local and global categories to the user. The user selects one and presses button 'Set Filter' to set the current search filter. The next page shown will be the according submenu with the current search filter applied. Pressing 'Cancel' leads back to the search filter generation page.

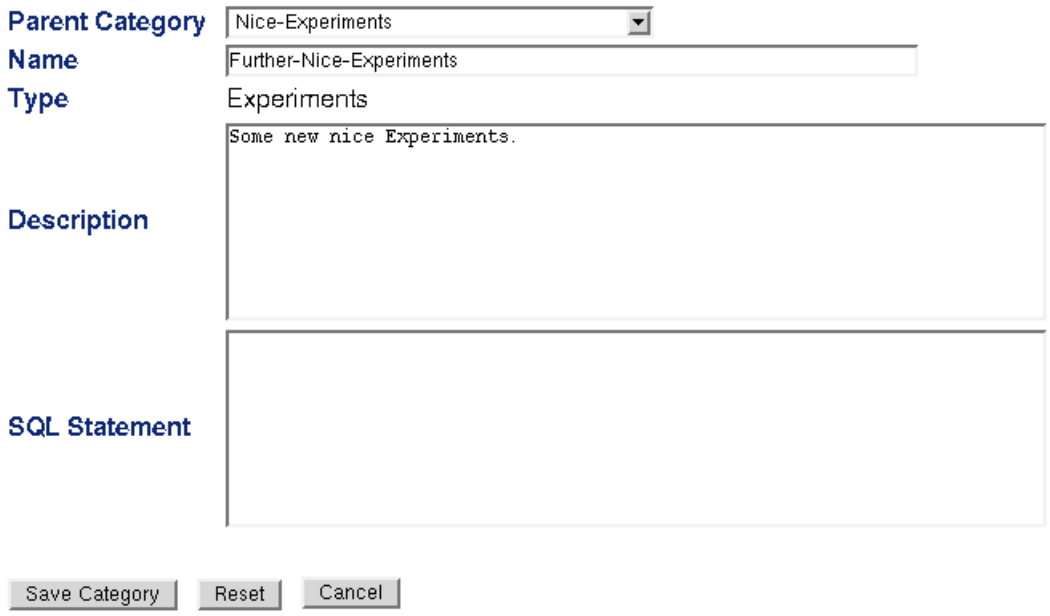


Figure 3.43: Add a category for experiments

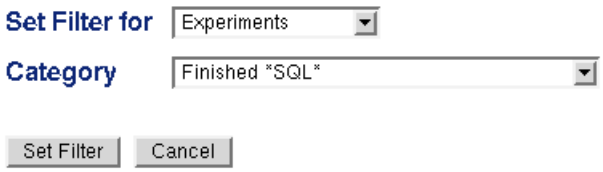


Figure 3.44: Setting current search filter from categories

Static categories can be used to define very specific data sets of problem instances, experiments or configurations only. Other types of objects are not supported yet for use in static categories. Additionally, local static categories can only be assigned static elements of the same type. In submenu 'Assign Categories' in submenu 'Search Filters' (see figure 3.45 on the following page), objects can be added statically to a static category, i.e. a category without a defining SQL statement. It is not possible to assign entries to an existing dynamic category. The type of object to add to a static category can be selected in selection box named 'Type'. Next, the user chooses the category to extend in selection box 'Category'. The categories that are available here will dependent on the object type chosen. If no object type has been chosen yet, only global static categories will be available. The next field is labeled according to the object type chosen in field 'Type' and provides a list of all objects of this type in the testbed. If no type has been chosen yet, it will be empty. By highlighting objects and subsequently pressing button button 'Set Category', the user can add the selected objects statically and permanently to the category. Multiple entries are assigned by holding down key 'Control/Ctrl' while selecting the entries. Since the list of objects of a given type contained in the testbed can be huge, input field labeled 'Filter XYZ' with 'XYZ' being the name of the object

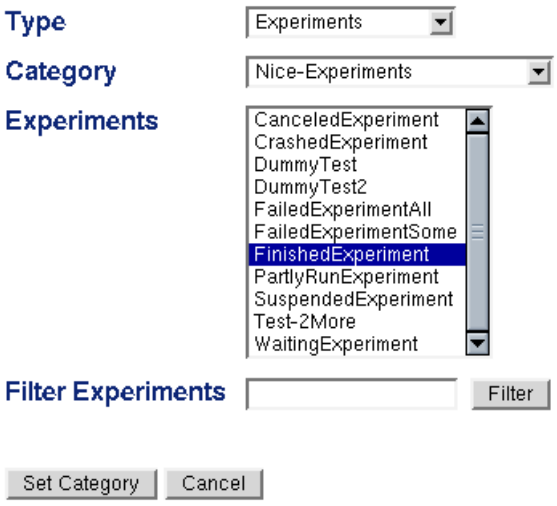


Figure 3.45: Assigning objects to categories

type chosen (or 'Available' if none has been chosen yet) provides means to reduce this number. The user can enter a regular expression as described in paragraph 'Wildcards and Regular Expressions' on page 162 that will leave only those objects, whose name matches the regular expression entered. Pressing button 'Cancel' leads back to the search filter generation mask.

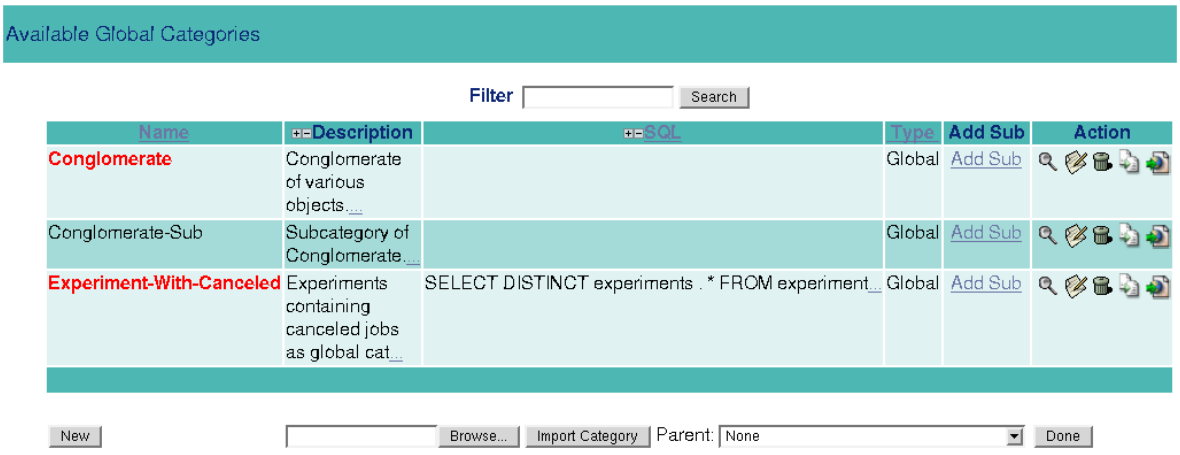


Figure 3.46: Managing global categories

Global categories are managed in their own submenu called 'Global Categories' (see figure 3.46) and can be static or dynamic, too. The advantage of a static global category is that objects of different types can be grouped together. They can not be viewed together in a detailed or other view yet, but if selecting a global category as category filter in a submenu, all objects of the submenu's type encompassed by the category will be displayed. Identifiers of dynamic categories in selection lists end with suffix `*SQL*`, while global categories end with suffix `<Global>`.

4 Advanced Topics

This section describes how to extend the testbed. This can be done in various ways. First of all, new modules can be integrated. These need to have a wrapper called module definition file as was explained in paragraph 2.3.1 on page 15 in subsection 2.3.1 on page 14. Next, the testbed can be extended by writing new data extraction and analysis scripts implementing a wide variety of methods to extract and subsequently analyze the results of jobs. For example, arbitrary plots, statistical tests, regression and the like can be implemented in the R language which is employed by the testbed to implement statistical analysis.

One is concerned with all details about integrating a module into the testbed. This section mainly explains how module definition files are constructed how to adjust them. The next section introduces the data extraction macro language used to flexibly write data extraction scripts, while the subsequent section treats the creation of analysis scripts. This section, however, is not an introduction into the R programming language. Finally, one short section describing how to install a web base interface for the testbed PostgreSQL database follows. The last section, then, is a list of troubleshooting and other general hints. Several basic errors and mistakes and their remedy are covered there. In case a problem arises, it is always a good idea to have a look at the troubleshooting section first. Additionally, it explains some peculiarities of the testbed in more detail than was adequate before.

Most parts of the testbed are written in PHP. A lot of variable parts of the testbed such as the wrappers for registering and running the executables of modules and the scripts for extracting data from results of experiments are using PHP, too. For this reason, some knowledge about PHP-programming is needed. Basic information about PHP-programming can be found in the PHP Tutorial on *http://www.php.net/tut.php* [55] and the PHP Manual [54] on *http://ctdp.tripod.com/independent/web/php/intro/index.html*.

Before plunging into the details of integrating modules into the testbed, and writing scripts, however, a brief introduction to PHP is given.

4.1 Quick Introduction to PHP

The following excerpts are taken from the PHP-manual ([54]) describing briefly the most important aspects of PHP:

Variables

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive. Variable names follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

Expressions and Operators

The most basic forms of expressions are constants and variables. When you type "$a = 5", you're assigning '5' into $a. '5', obviously, has the value 5, or in other words '5' is an expression with the value of 5 (in this case, '5' is an integer constant). After this assignment, you'd expect $a's value to be 5 as well, so if you wrote $b = $a, you'd expect it to behave just as if you wrote $b = 5. In other words, $a is an expression with the value of 5 as well. If everything works right, this is exactly what will happen.
The basic assignment operator is "=". Your first inclination might be to think of this as "equal to". Don't. It really means that the the left operand gets set to the value of the expression on the rights (that is, "gets set to").
The value of an assignment expression is the value assigned. That is, the value of "$a = 3" is 3. This allows you to do some tricky things:

```
$a = ($b = 4) + 5; // $a is equal to 9 now, and $b has been set to 4.
```

In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic and string operators that allow you to use a value in an expression and then set its value to the result of that expression. For example:

```
$a = 3;
$a += 5; // sets $a to 8, as if we had said: $a = $a + 5;
$b = "Hello ";
$b .= "There!"; // sets $b to "Hello There!", just like $b = $b . "There!";
```

Arithmetic Operators

Example	Name	Result
$a + $b	Addition	Sum of $a and $b.
$a - $b	Subtraction	Difference of $a and $b.
$a * $b	Multiplication	Product of $a and $b.
$a / $b	Division	Quotient of $a and $b.
$a % $b	Modulus	Remainder of $a divided by $b.

Comparison Operators

Example	Name	Result
$a == $b	Equal	TRUE if $a is equal to $b.
$a === $b	Identical	TRUE if $a is equal to $b.
$a != $b	Not equal	TRUE if $a is not equal to $b.
$a <> $b	Not equal	TRUE if $a is not equal to $b.
$a !== $b	Not identical	TRUE if $a is not equal to $b.
$a < $b	Less than	TRUE if $a is strictly less than $b.
$a > $b	Greater than	TRUE if $a is strictly greater than $b.
$a <= $b	Less than or equal to	TRUE if $a is less than or equal to $b.
$a >= $b	Greater than or equal to	TRUE if $a is greater than or equal to $b.

Logical Operators

Example	Name	Result
$a and $	bAnd	TRUE if both $a and $b are TRUE.
$a or $b	Or	TRUE if either $a or $b is TRUE.
$a xor $	bXor	TRUE if either $a or $b is TRUE, but not both.
! $a	Not	TRUE if $a is not TRUE.
$a && $b	And	TRUE if both $a and $b are TRUE.
$a || $b	Or	TRUE if either $a or $b is TRUE.

Strings

A string literal can be specified in three different ways.

- single quoted
- double quoted
- heredoc syntax

If the string is enclosed in double-quotes (”), PHP understands more escape sequences for special characters. When a string is specified in double quotes or with heredoc, variables are parsed within it.
There are two types of syntax, a simple one and a complex one. The simple syntax is the most common and convenient, it provides a way to parse a variable, an array value, or an object property.
If a dollar sign ($) is encountered, the parser will greedily take as much tokens as possible to form a valid variable name. Enclose the variable name in curly braces if you want to explicitly specify the end of the name.

```
$beer = 'Heineken';
echo "$beer's taste is great"; // works, "'" is an invalid character \
                                                          for varnames
echo "He drank some $beers";   // won't work, 's' is a valid character \
                                                           for varnames
echo "He drank some ${beer}s"; // works
```

Similarly, you can also have an array index or an object property parsed. With array indices, the closing square bracket (]) marks the end of the index. For object properties the same rules apply as to simple variables, though with object properties there doesn't exist a trick like the one with variables.

```
$fruits = array('strawberry' => 'red', 'banana' => 'yellow');

// note that this works differently outside string-quotes
echo "A banana is $fruits[banana].";
echo "This square is $square->width meters broad.";

// Won't work. For a solution, see the complex syntax.
echo "This square is $square->width00 centimeters broad.";
```

For anything more complex, you should use the complex syntax. Complex (curly) syntax This isn't called complex because the syntax is complex, but because you can include complex expressions this way. In fact, you can include any value that is in the namespace in strings with this syntax. You simply write the expression the same way as you would outside the string, and then include it in { and }. Since you can't escape '{', this syntax will only be recognised when the $ is immediately following the {. (Use ”{\$” or ”\{$” to get a literal ”{$”). Some examples to make it clear:

```
$great = 'fantastic';
echo "This is { $great}"; // won't work, outputs: This is { fantastic}
echo "This is {$great}";  // works, outputs: This is fantastic
echo "This square is {$square->width}00 centimeters broad.";
echo "This works: {$arr[4][3]}";
```

```
// This is wrong for the same reason
// as $foo[bar] is wrong outside a string.
echo "This is wrong: {$arr[foo][3]}";

echo "You should do it this way: {$arr['foo'][3]}";
echo "You can even write {$obj->values[3]->name}";
echo "This is the value of the var named $name: {${$name}}";
```

Strings may be concatenated using the '.' (dot) operator. Note that the '+' (addition) operator will not work for this. Please see String operators for more information. There are two string operators. The first is the concatenation operator ('.'), which returns the concatenation of its right and left arguments. The second is the concatenating assignment operator ('.='), which appends the argument on the right side to the argument on the left side. Please read Assignment Operators for more information.

```
$a = "Hello ";
$b = $a . "World!"; // now $b contains "Hello World!"

$a = "Hello ";
$a .= "World!";     // now $a contains "Hello World!"
```

Arrays

An array in PHP is actually an ordered map. A map is a type that maps values to keys. This type is optimized in several ways, so you can use it as a real array, or a list (vector), hashtable (which is an implementation of a map), dictionary, collection, stack, queue and probably more. Because you can have another PHP-array as a value, you can also quite easily simulate trees. An array can be created by the array() language-construct. It takes a certain number of comma-separated key => value pairs.

```
array( [key =>] value
     , ...
     )
// key is either string or nonnegative integer
// value can be anything

array("foo" => "bar", 12 => true);
```

A key is either an integer or a string. If a key is the standard representation of an integer, it will be interpreted as such (i.e. "8" will be interpreted as 8, while "08" will be interpreted as "08"). There are no different indexed and associative array types in PHP, there is only one array type, which can both contain integer and string indices.

A value can be of any PHP type.

```
array("somearray" => array(6 => 5, 13 => 9, "a" => 43));
```

If you omit a key, the maximum of the integer-indices is taken, and the new key will be that maximum + 1. As integers can be negative, this is also true for negative indices. Having e.g. the highest index being -6 will result in being -5 the new key. If no integer-indices exist yet, the key will be 0 (zero). If you specify a key that already has a value assigned to it, that value will be overwritten.

```
// This array is the same as ...
array(5 => 43, 32, 56, "b" => 12);
// ...this array
array(5 => 43, 6 => 32, 7 => 56, "b" => 12);
```

Using `TRUE` as a key will evalute to integer 1 as key. Using FALSE as a key will evalute to integer 0 as key. Using NULL as a key will evaluate to an empty string. Using an emptry string as key will create (or overwrite) a key with an empty string and its value, it is not the same as using empty brackets. You cannot use arrays or objects as keys. Doing so will result in a warning: Illegal offset type. You can also modify an existing array, by explicitly setting values in it. This is done by assigning values to the array while specifying the key in brackets. You can also omit the key, add an empty pair of brackets ("[]") to the variable-name in that case.

```
$arr[key] = value;
$arr[] = value;
// key is either string or nonnegative integer
// value can be anything
```

If $arr doesn't exist yet, it will be created. So this is also an alternative way to specify an array. To change a certain value, just assign a new value to an element specified with its key. If you want to remove a key/value pair, you need to unset() it. Common operation with arrays are listed on the next page:

Name	Description
`array_count_values()`	Counts all the values of an array
`array_key_exists()`	Checks if the given key or index exists in the array
`array_keys()`	Return all the keys of an array
`array_search()`	Searches the array for a given value, returns its key if success
`in_array()`	Return `TRUE` if a value exists in an array
`sort()`	Sort an array
`arsort()`	Sort an array in reverse order and maintain index association
`asort()`	Sort an array and maintain index association
`krsort()`	Sort an array by key in reverse order
`ksort()`	Sort an array by key
`count()`	Count elements in a variable
`sizeof()`	Get the number of elements in variable
`key()`	Fetch a key from an associative array
`print_r`	Print the array

Booleans

This is the easiest type. A boolean expresses a truth value. It can be either `TRUE` or`FALSE`.
When converting to boolean, the following values are considered `FALSE`:

- the boolean `FALSE` itself
- the integer 0 (zero)
- the float 0.0 (zero)
- the empty string, and the string "0"
- an array with zero elements

Control structures

Any PHP script is built out of a series of statements. A statement can be an assignment, a function call, a loop, a conditional statement of even a statement that does nothing (an empty statement). Statements usually end with a semicolon. In addition, statements can be grouped into a statement-group by encapsulating a group of statements with curly braces. A statement-group is a statement by itself as well. The various statement types are described in this chapter.

if

The `if` construct is one of the most important features of many languages, PHP included. It allows for conditional execution of code fragments. PHP features an `if` structure that is similar to that of C:

```
if (expr)
    statement
```

Often you'd want to have more than one statement to be executed conditionally. Of course, there's no need to wrap each statement with an if clause. Instead, you can group several statements into a statement group. For example, this code would display $a is bigger than $b if $a is bigger than $b, and would then assign the value of $a into $b:

```
if ($a > $b) {
    print "a is bigger than b";
    $b = $a;
}
```

`if` statements can be nested indefinitely within other `if` statements, which provides you with complete flexibility for conditional execution of the various parts of your program.

else

Often you'd want to execute a statement if a certain condition is met, and a different statement if the condition is not met. This is what `else` is for. else extends an if statement to execute a statement in case the expression in the `if` statement evaluates to `FALSE`. For example, the following code would display a is bigger than b if $a is bigger than $b, and $a is NOT bigger than $b otherwise:

```
if ($a > $b) {
    print "a is bigger than b";
} else {
    print "a is NOT bigger than b";
}
```

The `else` statement is only executed if the if expression evaluated to `FALSE`, and if there were any `elseif` expressions - only if they evaluated to `FALSE` as well (see `elseif`).

elseif

`elseif`, as its name suggests, is a combination of `if` and `else`. Like else, it extends an if statement to execute a different statement in case the original `if` expression evaluates to `FALSE`. However, unlike else, it will execute that alternative expression only if the `elseif` conditional expression evaluates to `TRUE`. For example, the following code would display $a is bigger than $b, a equal to $b or a is smaller than $b:

```
if ($a > $b) {
    print "a is bigger than b";
} elseif ($a == $b) {
    print "a is equal to b";
} else {
    print "a is smaller than b";
}
```

There may be several `elseif`s within the same if statement. The first `elseif` expression (if any) that evaluates to `TRUE` would be executed. In PHP, you can also write '`else if`' (in two words) and the behavior would be identical to the one of '`elseif`' (in a single word). The syntactic meaning is slightly different (if you're familiar with C, this is the same behavior) but the bottom line is that both would result in exactly the same behavior. The `elseif` statement is only executed if the preceding if expression and any preceding elseif expressions evaluated to `FALSE`, and the current `elseif` expression evaluated to `TRUE`.

while

`while` loops are the simplest type of loop in PHP. They behave just like their C counterparts. The basic form of a while statement is:

```
while (expr) statement
```

The meaning of a while statement is simple. It tells PHP to execute the nested statement(s) repeatedly, as long as the while expression evaluates to `TRUE`. The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time PHP runs the statements in the loop is one iteration). Sometimes, if the while expression evaluates to `FALSE` from the very beginning, the nested statement(s) won't even be run once. Like with the if statement, you can group multiple statements within the same while loop by surrounding a group of statements with curly braces, or by using the alternate syntax:

```
while (expr): statement ... endwhile;
```

for

`for` loops are the most complex loops in PHP. They behave like their C counterparts. The syntax of a for loop is:

```
for (expr1; expr2; expr3) statement
```

The first expression (`expr1`) is evaluated (executed) once unconditionally at the beginning of the loop. In the beginning of each iteration, `expr2` is evaluated. If it evaluates to `TRUE`, the loop continues and the nested statement(s) are executed. If it evaluates to

FALSE, the execution of the loop ends. At the end of each iteration, expr3 is evaluated (executed). Each of the expressions can be empty. expr2 being empty means the loop should be run indefinitely (PHP implicitly considers it as TRUE, like C). This may not be as useless as you might think, since often you'd want to end the loop using a conditional break statement instead of using the for truth expression.

foreach

PHP 4 (not PHP 3) includes a foreach construct, much like Perl and some other languages. This simply gives an easy way to iterate over arrays. foreach works only on arrays, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variables. There are two syntaxes; the second is a minor but useful extension of the first:

```
foreach(array_expression as $value) statement
foreach(array_expression as $key => $value) statement
```

The first form loops over the array given by array_expression. On each loop, the value of the current element is assigned to $value and the internal array pointer is advanced by one (so on the next loop, you'll be looking at the next element). The second form does the same thing, except that the current element's key will be assigned to the variable $key on each loop.

break

break ends execution of the current for, foreach, while, do..while or switch structure. break accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of.

continue

continue is used within looping structures to skip the rest of the current loop iteration and continue execution at the beginning of the next iteration. continue accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of.

4.2 Integrating Modules into the Testbed

The testbed in the form of a job server eventually executes the module binary executables. However, a job server does not address it directly. Instead, a wrapper written in PHP is needed for each module that is supposed to be integrated into the testbed. This wrapper is called the module definition file and will register the module's parameters to the testbed and eventually will execute any binary executable. A job server communicates with the executables only via module definition files. Modules can be integrated into the testbed simply by placing the executable and the module definition file for the testbed in the appropriate directories and registering the module definition file to the testbed. The testbed will only see the module definition file directly with its description of the interface module. Note that the module definition file must always have access to the particular executable. Each executable that heeds the optional interface requirements as defined in subsection 2.3.1 on page 15 can be registered easily as a module to the testbed. If a module does not heed the command line interface definition format of the testbed (compare to subsection 2.3.1 on page 15), it more complicated to produce an appropriate module definition file. In case the format is respected, a module definition file can be generated completely automatically.

This subsection explains how to write or generate a module definition file and which settings might have to be changed manually in order to get a module definition file to work with both the testbed and the executable.

When a module definition file was generated, the module can be registered to the testbed and thus integrated with CLI command

```
testbed module register <modulename>
```

The module definition file and the executable should be in the appropriate locations (see 3.2.2 on page 76). A module can be removed from the testbed with command

```
testbed module remove <modulename>
```

issued on the CLI. Note that before a module can be removed, all algorithms, configurations, experiments, and so on using the module have to be removed first. For complementing information about the topic compare to subsections 3.4.2 and 3.2.2 on pages 138 and 76, respectively.

Note: The examples presented in this subsection are located in directory

`DOC_DIR`*/examples/modules/*.

4.2.1 Module Definition File Generation Tools

Two tools for automatic and semi-automatic generation of a module definition file from the command line interface definition output of an executable exist. These tools can

be found in directory `TESTBED_ROOT`*/devel/*. They are named *gen_module_from_mhs.php* and *gen_module.php*. Additionally, these tools can be addressed by commands

```
testbed modules makeConform
```

and

```
testbed modules makeNonConform
```

respectively.

If the executable is conform to the command line interface definition format as defined in subsection 2.3.1 on page 24, tool *gen_module_from_mhs.php* can be used to create a module definition file completely automatically. For any other output format tool *gen_module.php* is used. Depending on the output of the executable when called with parameter `--help`, the resulting module definition file will have to be edited manually. For example, not all subrange restrictions from the command line interface definition output as described in subsection 2.3.1 on page 14 will be translated and used when setting parameters. Special subrange restrictions have to be added or refined manually in the module definition file.

Both tools are called with the executable of the module that is to be registered as first argument. Each tool will ask for some additional information such as the name of the module, its problem type, a description (which is prepended before any comments given in the modules command line interface definition output (compare to paragraph 'Parameter Specification of Modules' in subsection 2.3.1 on page 15), and internal parameters (internal parameters are addressed later). Note that the settings made for the internal parameters are always appended to the call to the executable at system level as are, i.e. they are taken literally. If they are erroneous, e.g. the parameter name is misspelled, the executable might complain and fail to execute. Check the job server output to console for the exact call on system level. Note also that tool *gen_module_from_mhs.php* removes any parameters set as internal parameters that are described also in the `begin/end parameters` section of the command line interface definition output from the final module definition file. This avoids errors by duplicate parameter calls. Any internal parameters that are to be removed on demand must be in proper long flag format, in particular they must have the two leading '-' characters, otherwise they will not be recognized as parameter flags. Parameters set in the internal parameters section will not show up in the testbed later except they are exported by the executable to its output file. In particular, they can not be configures in any way.

If successful, the module definition file generated is written to the current directory. It is recommended to check this newly generated module definition file first for correctness before integrating the module with its help to th testbed. Note that the module name that is entered is used to uniquely identify the module in the testbed, regardless what the executable's name is. An error will occur, if trying to register a module with the same name as a module already registered to the testbed. Note also that the name

entered can be arbitrary long, but only the first 32 characters are used for identification, though. The module name may only consist of characters 'a' - 'z', 'A' - 'Z' and '0' - '9'. Invalid character will be removed silently. The following is an example of how to use a module definition file generator:

```
#> TESTBED_ROOT/devel/gen_module_from_mhs.php example
Module Name: Example
Problem Type: Dummy
Description: Just an example.
Internal Parameters:
```

The command line interface format that is expected by tool *gen_module.php* is less restrictive than the standard testbed command line interface definition format. Each line of the interface output of the executable should start with a `--<longflagname>`, followed by a `-<shortflag-character>`. The rest of the line is taken as a description for the parameter. Note that for this weaker format, character `#` does not indicate the beginning of a comment! Additionally, the internal default values are not checked against the parameter definitions found. If there are duplicates, the executable might be called with two time the same parameter.

Table 4.1 presents an example of the `--help` output that *gen_module.php* expects. The example is available as a shell script in the examples directory and is named *WeakCLIDefinitionOutput*. The accordingly generated module definition file is named *module.WeakCLIDefinitionOutput.inc.php*.

```
--time     -t  Maximum runtime (in seconds)
--tabu     -l  Length of tabu list
--tInit    -m  Initial Temperature
--alpha    -a  Alpha value for annealing schedule
--optimal  -o  Stop when hitting a solution of high quality
--quality  -i  Value of hight quality
--input    -i  input file
--output   -o  Output file
```

Table 4.1: Example of a simple command line interface definition output

4.2.2 Basic Settings

After generation of a basic module definition file with any generation tool or by copying and editing an existing module definition file, it is recommended to check, if the settings in the module definition file are correct. Altogether, the module definition file must be in correct PHP syntax! The following basic settings should be verified also:

- **Module Name**

  In lines

  `class module_{modname} extends basemodule,`

  `function module_{modname}(),`

  and

  ```
  var $ModulDescription = array(
   'module' => '{modname}',
  ```

  Place Holder `{modname}` must always be the same name, otherwise the module can not be registered and executed. The module name may only consist of characters 'a' - 'z', 'A' - 'Z' and '0' - '9'. If generation tools are used, any other characters will be removed automatically and silently. If the module definition file is written manually, the user must ensure that no invalid characters are used. Otherwise, an error will occur.

- **Executable**

  In line

  `var $executable = '{binary}';`

  the correct name and path of the executable must be entered. If the executable is placed in its standard location, which is `TESTBED_BIN_DIR`/<*arch*>/<*os*>/<*modulename*>, only its name has to be given here. Place holders <*arch*>, <*os*>, <*modulename*> stand for the architecture, operating system and module specific subdirectories of the root directory for all binaries, respectively. For example, on a Linux system with a Pentium processor, <*arch*> is *i386* and <*os*> is *Linux*.

- **Description**

  This attribute of the module as contained in variable `var $ModulDescription` gives a brief description of the module to the user. A line

  `'description' => '{here comes the description}',`

  must be present somewhere in section `ModulDescription` of the module definition file. Place Holder `{here comes the description}` is the description of the module in the form of a string. It can be empty, of course. Note that the comma at the end of the line must not be omitted!

- **Problem Type**

  If the module is specialized for a specific problem type, a line

  `'problemtype' => '{shortcut of the problem type}',`

  somewhere in the section beginning with `ModulDescription` must be present (place holder `{shortcut of the problem type}` again is a string). Note again

that the comma at the end of the line must not be omitted! Which problem types already exist in the testbed can be viewed on the web front end. If the specified problem type does not already exist, it will automatically be created without any description. After registration of the module (see subsection 3.2.2 on page 76), the description of the new problem type can be set belatedly in the web front end (see subsection 3.3.3 on page 103). If the module can be used on different problem types, the previously mentioned line specifying the problem type can be removed or `'{shortcut of the problem type}'` can be set to an empty value `''`. That way, the module can be used for any problem type.

- **Internal Parameters**

  Sometimes, the user wishes to hide some of the command line parameters of a module completely from the testbed or set some parameters to its own default values different from module internal default values but transparently for any later usage in the testbed (compare with subsection 3.3.6 on page 107). This can be the case, for example, if some fixed parameter values are required for the executable to run correctly in batch mode. For example, a module executable might normally start with a graphical user interface (GUI), but fortunately, the GUI can be disabled with parameter flag `--console`. In cases like this, parameters can be set permanently with the following line in the module definition file:

  ```
  $this->InternalParameter ='{parameters}';
  ```

  This line must be contained in function `function module_{modname}()`. The parameters set here need not comply to the flag + value syntax of the CLI definition output format. As such, the section can be used to set parameters that consists of a flag only. The difference between internal parameters and parameters that are set to a default value when defining an algorithm with the help of the web front end is that the latter will be available when extracting data, while the former will not. Both, however, will be set when calling the executable on the CLI. If there is no need for special parameters, the value must be empty, but it is not allowed to remove the line! Parameters of the executable set to a default value in the internal parameters are always appended by the testbed when executing the executable of a module. Internal parameters should not be listed in the parameter section which is discussed next in case of tool *gen_module.php*, since in this case, the parameter could be used twice with unpredictable consequences. The consequences depend on the implementation of the module executable and may easily make the executable not work properly. In case of tool *gen_module_from_mhs.php*, parameters of the internal parameters section are removed automatically. Any internal parameters that are to be removed on demand must be in proper long flag format, in particular they must have the two leading '-' characters, otherwise they will not be recognized as parameter flags. When changing the module definition file later manually, such a removal must be ensured manually, too!

After checking all these sections a module definition file can look like the following these sections:

```
<?php
class module_Dummy extends basemodule
{
    /*  Name of binary executable. No need to specify a path here,
    **  if binary put in directory TESTBED_BIN_DIR/<arch>/<os>/<modulname>/.
    **  Otherwise use an absolute path starting with a /. */
    var $executable = 'Dummy';

    /*  Description of module. See user manual for a full list of
    **  featured attributes. */
    var $ModulDescription = array(
        'module' => 'Dummy',
        'problemtype' => 'Dummy',
        'description' => 'Dummy module for use with the testbed.
                          Used for testing and demonstration purposes.',
    );

    /*  Parameter description. See user manual for all featured attributes. */
    var $ParamDescription = array(
        'input' => array(
            'description'  => 'Input-file',
            'cmdline'      => '-i',
            'cmdlinelong'  => '--input',
            'typ'          => 'filename',
            'paramtype'    => 'FILENAME',
            'defaultvalue' => 'a.dat',
        ),

        . . .

        'randomY' => array(
            'description'  => 'Degree of randomization of measurements.',
            'cmdline'      => '-r',
            'cmdlinelong'  => '--randomY',
            'typ'          => 'real',
            'paramtype'    => 'REAL',
            'condition'    => '/^([+]?([0]*[1-9][0-9]*(\.[0-9]+)?|
                               [0]*\.[0]*[1-9][0-9]*)|[+-]?([0]+(\.[0]+)?|\.[0]+))$/',
            'paramrange'   => '>=0',
            'defaultvalue' => '1',
        ),
    );
```

```
    /*      Description of module's performance measures, if run as
    **      the last one. See user manual for a full list of
    **      featured attributes.  */
    var $PerformanceMeasure = array(
         array(
             'name' => 'best',
             'type' => 'REAL',
         ),

         .  .  .

         array(
             'name' => 'stepsWorst',
             'type' => 'INT',
         ),
    );

    function module_Dummy()
    {
        $this->InternalParameter ='';
    }

    /*  Do not change anything below.
    **  Change only, if change of execution mode is necessary. */
}
```

4.2.3 Parameter Definition

After the basic settings were made, the parameters, as supported by the module, can be defined. Parameters have attributes assigned such as short and long flags, a type and subrange, a default value, and a description. Each parameter has its own entry in the module definition file. The syntax is that of an array in PHP. The name of the variable that holds the defining array for each parameter will become the name of the parameter. This name is used to construct unique parameter names when creating an algorithm as described in subsection 3.3.6 on page 110. The elements of the array will become the attributes of the parameter. The parameter name may only consist of characters 'a' - 'z', 'A' - 'Z' and '0' - '9', all other characters are not allowed. If the definition file is written by the user, the user must ensure that no invalid characters are used.

The following list presents and describes the attributes of parameters. If an unknown attribute is used, the registration of that module will fail with a database error. At least the short or long flag command line option for a parameter and its type attributes

(`paramtype` and `typ`) must be defined. The long flag definition will be preferred over the short flag definition when starting the executable.

- **description**

  A short description about the effect and purpose of the parameter.

- **cmdline**

  The short flag command line option for the parameter.

- **cmdlinelong**

  The long flag command line option for the parameter.

- **paramtype**

  Type of values the parameter can accept. The following types are known:

  - **FILENAME**: A file name is expected as input, encoded as a string. The filename must contain path information when given through configurations of the testbed (see subsection 3.3.7 on page 110 and paragraph 2.3.1 on page 15), otherwise the files will not be found since it they are looked for in a temporary directory created by the testbed on demand.
  - **BOOL**: The parameter value will either be `1` representing true or `0` representing false.
  - **STRING**: Any string of characters is a valid parameter value.
  - **INT**, **REAL**: The parameter value must represent an integer or real number in conventional floating point notation, respectively [1]

  The type information will be presented together with the subrange restriction in column named 'Type' to the user when parameters of a module are to be set (compare with subsection 3.3.6 on page 107 and subsection 3.3.7 on page 110). However, the value of this column itself does not trigger a range checking for values entered by the user. Any validity checking of parameter settings is controlled by parameter attribute **condition**. This attribute contains a regular expression that does exactly this.

- **typ**[2*]

  This attribute contains the same information as attribute **paramtype**, only the type names *must* be all *lower case*. This attribute is for internal usage only.

[1] PHP does not have strict data types like C, instead all data types are treated as strings and are converted on demand to numeric data types.

[2] Type as written in German

- **defaultvalue**

  If a parameter is not set by the testbed, the module will internally use this default value. Note that the module will use its internal default value, even if the internal default value is different from a false value misleadingly set here. The value set here is of pure informative nature. It even need not be valid with respect to the type and subrange settings.

- **paramrange**

  Subrange for parameter types such as `:<0`, `:(1,2]`, `:>=500`, `:{0.1,0.3,1.5}`. This information will presented together with the type information to the user when defining an algorithm or a configuration.

- **condition**

  A regular expression defines which parameter values will be accepted as input from the user. If the module definition file was generated with tool *gen_module_from_mhs.php*, the type information (and most subrange information) from the command line interface definition output of the module for this parameter will be used to generate a regular expression. This regular expression is used to check whether an actual setting of this parameter, i.e. whether the setting can be interpreted as a value of the subrange of the type. Regular expressions for complex numerical intervals are not generated automatically. Complex numerical intervals are only translated to a regular expression checking the type restriction. In particular, any open, closed, or half-open intervals are only used to check any given default value but are not used to check for proper user input later. Additionally, only $<= x$, $<= x$, $> x$, $>= x$ with $x = 0$ will be translated. If additional subrange for complex numerical intervals is needed, the user has to modify the regular expression manually here. If the regular expression entered here does not work properly, an error will occur later when using the web front end. Besides, the regular expression entered or modified here manually might not be conform with the type and subrange information of the command line interface definition output of the module and consequently will not be conform with the information given for the parameter when setting a parameter value. For more information about forming proper regular expressions see the PHP-manual 'Regular Expression (Perl Compatible), Pattern syntax' ([54]).

  Note: In paragraph 'Parameter Subrange Checking' on page 274 in chapter 6 on page 271, two utility programs for automatic generation of regular expressions for intervals of real values are described. The tools have not completely tested yet, though.

In a nutshell, a parameter definition can look like this:

```
'tries' => array(
        'description'  => 'Number of tries (=repetitions) of algorithm',
```

```
        'cmdline'      => '-x',
        'cmdlinelong'  => '--tries',
        'typ'          => 'int',
        'paramtype'    => 'INT',
        'condition'    => '/^[+]?[0]*[1-9][0-9]*$/',
        'paramrange'   => '>0',
        'defaultvalue' => '10',
),
```

4.2.4 Defining Performance Measures

For the statistical interpretation of experimental results it is crucial to know which performance measures a module provides if it was run last in an algorithm. Typically, an experimenter is interested in the response of the algorithm to different parameter settings, input files, or conditions with respect to one or more performance measures.

Again, the syntax to define performance measures is adopted from PHP. The performance measures are represented in the module definition file as an array of individual performance measures, which are represented as an array of key value pairs. The name of the performance measure is indicated by key `name` and must always be specified. The type of a performance measure is indicated by key `type`. Further attributes of an performance measure are not supported, yet. There can be more than one performance measure for the last module. A definition of two performance measure could look like this:

```
var $PerformanceMeasure = array(
    array(
        'name' => 'best'
        'type' => 'INT'
    ),
    array(
        'name' => 'length'
        'type' => 'INT'
    ),
);
```

Note that the order of the performance measures is irrelevant.

4.2.5 Adjusting the Execution Part

If the executable implementing a module does not support the minimal set of parameters required by the testbed command line interface definition format as defined in table 2.1 on page 16, the execution part of the module definition file can be adjusted to make

it work with the testbed nevertheless without employing an additional wrapper. For example, if the module does not support the `--output` flag (file to write the result of the module) but unalterably write its results to standard output, the module definition file can be extended to automatically redirect the standard output of the executable to a file whose name is given by the output parameter by the testbed. At the moment there is no possibility to specify more than one output file. Due to this restrictions additional output data in separate files can not be stored back to the database. Furthermore, after running a job the output to standard output or standard error can not be stored automatically in the database either. If the module is not run as the last module of an algorithm, the execution part of the module can be adopted to pack more files into one. The next module then has to unpack these files again. In this case, some bigger modifications of the execution part have to be dealt with. In case of an example with more than one output file, both the execution parts of the module definition file for the module packing two files into one and for the modules unpacking two files have to be augmented by such a packing and unpacking mechanism. In case the output file parameter flag `--output` simply is named differently compared to the command line interface definition format, the module definition file can translate the parameter. Module *module.lsmcqap.inc.php* found in the examples directory was adjusted that way. The source code can also be found in appendix A.1.1 on page 283.

Another way to make a module executable heed the parameter interface restrictions posed by the testbed is to write an additional wrapper for the executable which then is viewed as the module executable by the module definition file. Hence, in this case the module has two wrappers assigned. A shell wrapper for an executable that does not support any of the testbed required standards is shown in appendix A.1.2 on page 286. This shell wrapper makes the executable run inside the testbed with all required standards. The changes could also be made in the PHP source of the module definition file. Note, however, that this is not easy in this case, because the execution part, crucial for running a executable by the module definition file, would have to be modified substantially which is not an easy task for an unexperienced user.

While changes to the executable can be conducted and the executable can be replaced at its appropriate location if its command line interface remains the same, module definition files have to registered again, if they have changed. Be aware that a new registration under the same name is only possible, if the corresponding old module has been removed from the testbed before which, in turn, is only possible, if all objects depending on this module, most of all algorithms, have been removed, too. One way to do this anyway, without loosing data, is by exporting and re importing the dependent objects to XML.

4.3 Writing Data Extraction Scripts

This subsection explains how to write scripts for extracting data from algorithm output that can exploit the standard output format of the testbed (compare to subsection 2.3.1 on page 24). Such, data extraction scripts constitute one of the pivotal interfaces as identified in section 2.2 on page 11 and depicted in figure 2.2 on page 12. Data extraction scripts are used to extract data from the result of jobs. Recall that the result of job is the output file written by the last module of the job's algorithm containing the information about the results of the run. The degree of automation in data extraction with the testbed rises with increased conformity of job results with the testbed's standard output format as defined in subsection 2.3.1 on page 24. Data extraction scripts, or extraction scripts in short, scan the results of jobs, extract certain information and provide the information extracted as tables of data in a way similar to tables in relational databases. Such a table consists of a list of data sets (also called *lines* or *rows*), each data set having the same number of fields, each such field consisting of a name value pair with the value possibly being empty. The values of a field over all lines can be regarded as the columns of the table. In this form, data extracted can easily be exported and next input to and processed by the R statistics package [60] or a plotting program such as Gnu-plot. This is done by exporting the data to a file with one line for each data set and the field values for each data set separated by a special character, for example a comma. Such files are called CSV files , which can be read by R.

The data extraction scripting language essentially is the PHP programming language: Each data extraction script essentially is a small PHP program. In order to simplify the construction of extraction scripts, a small set of commands and predefined variables with reserved names has been developed which helps in automating the most common extraction procedures for job results based on the testbed standard output format. The commands control which parts of the raw data from the job result are to be extracted. Raw data denotes the data of the job result output file in unprocessed form. Portions of the raw data that are extracted by some commands include, for example, performance measure values, parameter settings, solution encodings, and so on. Some parts of the raw data are always extracted and will be provided by predefined variables. Predefined variables as well provide information not contained in any job result such as parameter settings as used by the testbed. All other parts of the raw data typically is extracted by functions which provide the extraction results as return value in the form of PHP data structures. The data extracted can then be further processed with PHP constructs before the final result table of the extraction effort is constructed and output.

When writing extraction scripts the user is not confined to use only predefined commands of the data extraction language, but can also use any function of PHP. In particular, arbitrary data structures can be build and loops can be performed to further process the parts of the raw data that were extracted with predefined commands. For more information about PHP see the PHP tutorial [54]. The constructs of the data extraction

language are macros that are textually substituted as is done, for example, by the C programming language preprocessor. This should be kept in mind when writing extractions scripts, for sometimes the commands used might not lead to the expected result because of a wrong understanding of exactly what is substituted by what.

In general, an extraction script is applied to the result of each job in as set of jobs. As described in subsection 3.3.10 on page 125 these sets of jobs typically are the results of search filters, either in the form of a current search filter, a category or a predefined search filter for an experiment. The extraction process can be described as follows. First, usually with the help of predefined commands, for each job some parts of the raw data of the job's result are extracted from the job result. This data is, possibly after some processing and computing with PHP constructs, stored in an intermediate table. After all data has been extracted, processed and added to this intermediate table it is finally transformed into a new format which still is in table form. It is then extended with other information available for the job like parameters used together with their values, the job's experiment, and configuration and so on as known by the testbed by attaching this data to the transformed table, too. The extended tables, one for each job in the set of jobs that is processed, are then put together to form the final result of the data extraction script application in the form of a single table.

In what follows, the different table formats of the data extraction process, the commands and predefined variables of the testbed's data extraction language are described. First, the different phases of the data extraction process are described in terms of the format for the intermediate tables. Next, the commands and predefined variables are discussed. Subsequently, some examples are presented before a concluding paragraph provides some more detailed information about writing data extraction script such as common mistakes, troubleshooting assistance, and hints.

The examples presented in this subsection are available as XML exports of extraction scripts. The XML files are located in directory `DOC_DIR`*/scripts/extraction*. Recall that data extraction scripts are always named **.X.xml*.

4.3.1 Table Format

In order to better understand how data extraction scripts work, it is useful to have a closer look at the format the data has in the different stages during the extraction process and which format finally will be output and subsequently has to be read by a statistics package. In this respect, the table formats described here explain the third of the four central interfaces as identified in the testbed requirements subsection (subsection 2.3.1 on page 24, depicted in figure 2.2 on page 12). The data formats for the different stages of data extraction are as follows:

- Recall that the data in the standard output format is divided into blocks, lines

and fields, fields again are subdivided into a name and a value component. Lines are the atomic parts of each job result (compare with subsection 2.3.1 on page 14 on page 24). Hence, data is extracted on a line by line basis. Each line will be provided as a list of name value pairs in the form of an array. The length of this array, i.e. the number and types of fields per line can vary.

- These lines can be processed by some PHP statements and are then stored together in a table which is accessed through a predefined variable with reserved name `$result`. The table accessed through `$result` is represented as an array of lines, in the form of an array of arrays. Note that since this table is represented as an array of arrays, the columns of the table, i.e. individual fields, can not be accessed directly at this stage; only lines can be accessed directly. In addition, each line need not to have the same number and kinds of fields. Array `$result` constitutes the intermediate table of the extraction process.

- In the last stage, additional job specific information is added as well as performing a reformatting of the data extracted and processed data as contained in table `$result`. These operations typically are initiated with command `list()` which will be described later. Essentially, the table of variable `$result` in the form of a list of lines now becomes a list of fields, again in the form of an array of arrays, the inner arrays now representing a column of the table, each column corresponding uniquely to one field. If a line does not contain a certain field, the entry for this line in the corresponding column of the field will be empty. The job specific information is added in new fields, i.e. columns. Since this information is valid for all lines it is repeatedly stored for each line in the appropriate column. This conversion process can be viewed as transposing the table stored by variable `$result` and adding new columns to the transpose. This whole reformatting is conducted to better merge the individual results of the sets of jobs that is processed. The resulting, i.e. transposed, table is stored in reserved variable `$retval`.

- In the end, the data extracted for each job will be a table whose columns are named after all different kinds of information of fields as found or computed during the data extraction process including information about the job such as parameter settings. Each line of this table corresponds to one piece of coherent information found in a job result. A line will not contain information with respect to a certain field or column, if no such field was extracted together with this piece of information, i.e. line. The tables of all jobs of a set of jobs processed are now merged by appending them. These tables can have different sets of fields or columns. Any field from any table is again represented by a column in the merged table. If such a field was not known in a result table of a particular job, the column will be empty for all lines of this job's table. Altogether, the result of applying a data extraction script to a set of jobs, e.g a search result, is a table of data sets, each data set occupying one line consisting of a number of, possibly empty, fields. For example,

for each of a number of algorithms and a number of problem instances the data sets extracted could represent the points of solution quality vs. runtime trade-off curve which subsequently can be used to plot the curves.

4.3.2 Commands and Predefined Variables

The following commands, additional to all PHP commands, and the following predefined variables are available in the data extraction language. Predefined commands basically exist for all predefined blocks of the standard output format described in 2.3.1 on page 24 providing the information stored there in form of arrays as their return values. Note that comments are `//` for commenting out one single line and `/*` and `*/` for commenting out a region.

- `<!--userinput-->`, `<!--/userinput-->`

  In order to make extraction scripts more generic, some interactive user input is available. This is useful if an extraction script is supposed to be some kind of template. For example, if a script is supposed to extract only one performance measure of a number of performance measures as exported by the command line interface definition of the last module of an job's algorithm, the particular name of the performance measure can be requested interactively from the user before starting the script. This avoids pure copy and paste with extraction scripts, since these settings would otherwise have to be done in the script itself.

  Which user input is needed can be specified in a section bracketed by brackets `<!--userinput-->` and `<!--/userinput-->`. The specification of the user input requests is given in the form of a PHP array named `$userinput`. The elements of this array represent the different user input requests. The element names or rather keys will become the names of the variables that will be used in the script to store the user input (the variable names begin with an additional `$`, of course). Note that these variable names should not collide with any other predefined or any other variable used throughout the script. The elements of `$userinput`, arrays themselves, can have the following elements, i.e. key value pairs, indicated by their names:

  - `description`
    The value of this element is a description that is output left to the input field when the input request is presented to the user (compare with subsection 3.3.10 on page 125).
  - `type`
    This element's value indicates the type of the user input. Two kinds of user input are supported. If this entry is omitted the input requested will be a

string which can be input into a text input field. If the type is `selection`, a selection box will be shown to the user.

- `values`
  If the user input is supposed to be a selection box, the value of this element, being an array, can hold the entries of the selection box. Each key value pair represents one selection box entry. The value of such an entry will appear as a string in the selection box and can be clicked by the user. The keys are the values that will be stored in the variable accessing the user input. This variable will hold the key of the entry that was selected by the user.
- `default`
  In case of a selection box, this element's value contains the key of the selection box entry that is supposed to be the default value. The key's value is highlighted in the selection box when first presented to the user. If the user input requested is supposed to be string, the value of this element will already be written in the text input field when presented to the user. The user can edit this default string, of course.

Example:

```
<!--userinput-->
$userinput["ExampleSelList"] = array(
    "description" => "Example for user input via selection list:",
    "type" => "selection",
    "values" => array("A"=>"Type A","B" => "Type B", "C" => "Type C" ,
                      "D" =>"Type D"),
    "default" => "A"
);
$userinput["ExampleTextInput"] = array(
    "description" => "Example for textual user input:",
    "default" => "Default"
);
<!--/userinput-->
```

In this example, element named `ExampleTextInput` specifies a text input field, while element `ExampleSelInput` specifies a selection box. The values for these two user inputs can be accessed later in the script with variables `$ExampleSelInput` and `$ExampleTextInput`, respectively. The values for `$ExampleSelInput` will either be `A`, `B`, `C`, or `D`, the choice proposed to the user by default will be `A`. The user will see `Type A` in the selection box already selected and will have choices named `Type A - Type D`. The value for `$ExampleTextInput` will be the string the user enters into the text input field which is filled with string `Default` at the beginning. The example is available as XML export located in directory `DOC_DIR`*/scripts/extraction* named *Userinput-Example.X.xml*.

- `begineachtry{ , }endeachtry`

  Recall that the testbed requires provision of repeated independent runs of a job's algorithm on the job's problem instance called *tries* (see subsection 2.1.2 on page 7). The results of these tries are bracketed by `begin try #`, `end try #` with `#` being the number of the try as is demanded by the standard output format of the testbed. Each command inside brackets `begineachtry{` and `}endeachtry` now is applied to the results, i.e. lines in the try block of each try. There can be arbitrary blocks indicated by these brackets in the script. These must not, however, be nested or interleaved.

- `begineachrow{`, `}endeachrow`

  Each command inside the brackets `begineachrow{ , }endeachrow` is applied to each line of the try block that currently is processed. These brackets can only be used inside the `begineachtry{` and `}endeachtry` brackets. However, they can be used multiple times in one `begineachtry{` – `}endeachtry` block but must not be nested or interleave with other `begineachrow{ , }endeachrow` or `begineachtry{ , }endeachtry` brackets.

- `break`

  In order to leave a `begineachrow{`, `}endeachrow` or `begineachtry{ , }endeachtry` block prematurely, command `break` can be used. Using this command will leave the most inner block.

- `$row`, `$lastrow`

  While scanning the results of a job with the command in the `begineachtry{` – `}endeachtry` and `begineachrow{` – `}endeachrow` blocks, the basic information is eventually found separated into lines as required by the testbed standard output format. Accordingly, the result of a try block in the job result is scanned line by line. The results of the scan of the line that is currently processed, also called *current line*, will be stored in predefined variable `$row`. This variable is assigned a value for each line actually scanned. Not all lines will be scanned, but only those that contain *at least one* field with its name listed in variable `$perfMeasure` (see later). Recall that the structure of a line is a list of key value pairs separated by white space. This decomposition is performed regardless of the actual names and number of fields in the current line. It is not necessary that all lines have the same number and labeling of fields. Variable `$row` is an array, each element representing one field of the current line, the keys are the field names and the corresponding values are the field values of the line. For example, if a line of a job results look like

```
best 152906    cycle 31    steps 31    time 4.21    k_var 20
```

then variable `$row` will be an array of size six and each value of a field can be accessed with `$row["<fieldName>"]` in the script; `<fieldName>` can be `best`, `cycle`,`steps` `time`, or `k_var`. The value of field `best` is accessed with command `$row["best"]`. Additional to the fields from a line, which might differ from line to lone, the array in `$row` contains an element named `try` whose value is the number of the try block that is currently processed. This value is accessed with `$row["try"]`. Individual values calculated by a script can be added to the data structure `$row`, too, by just assigning them with a conventional array element assignment operator (see PHP manual [54]). That way, field values can be reassigned and new, derived fields can be created on scratch.

- `$lastrow`

  This variable works the same way as variable `$row` except that it holds the data from the last line that was scanned. If the first row of a try block is currently processed, this data structure contains no data, i.e. will be an empty array. After having ended to scan a try block this variable will hold the results of the last line scanned in this last try block, e.g. if accessed just after execution of a `begineachrow{` – `}endeachrow` block.

- `$try`

  This variable contains the number of the try block currently processed in a `begineachtry{` – `}endeachtry` block.

- `addresult(<dataset>)`

  As mentioned before, the results of scanning lines of try blocks will be stored in predefined variables `$row` and `$lastrow`. Before assigning a new value to these variables when scanning the next line, the old results should be stored. Command `addresult(<dataset>)` does exactly this. It adds any variable `<dataset>` to the internal evaluation structure hold by variable `$result` that stores all information extracted for later use with the `compute` or `list` commands as described later. Recall that each line can consist of several name value pairs. Hence, the variable added here must be an array of key value pairs. Typically, variables `$row` and `$lastrow` are entered with this command.

- `$perfMeasures`

  Variable `$perfMeasures` is an array of strings whose elements hold the names of a performance measures the script is supposed to extract. At the beginning of the script this variable will be set by default to hold the names of the performance measures as exported by the command line interface definition of the last module of the job's algorithm. Since variable `$perfMeasures` might have to be changed during the script, it is a good idea to save the contents in another variable before starting any other computation. When scanning the job result, only those

lines within a try block who have *at least one* field whose name is contained in variable `$perfMeasures` will be scanned, decomposed into its fields and the results stored in variable `$row` and hence in variable `$lastrow`. That is, variable `$perfMeasures` defines, which fields and hence which lines will be extracted and stored to variables `$row` and `$lastrow`. If `$perfMeasures` is empty, `best` is used as default performance measure. If more than one performance measure should be extracted, but only one at a time, variable `$perfMeasures` can be changed to have only one element at a time while running repeated `begineachtry{` – `}endeachtry` and `begineachrow{` – `}endeachrow` blocks. Note that the number of elements of any array (and hence the last index of variable `$perfMeasures`) can be obtained with PHP function `count(...)`.

- `$perfMeasureTypes`

  This variable is an array of the same size as the array stored in variable `$perfMeasures` at the beginning of a script. For each performance measure listed in `$perfMeasures` it contains the corresponding type in the form of a string in the same position. That is, the type information for the i-th performance measure listed in variable `$perfMeasures` is contained at position i of the array stored in `$perfMeasureTypes`.

- `PerfMeasureOut()`

  According to the standard output format, each job can output the performance measures it has employed and recorded in a special performance measures block in the job result enclosed in brackets `begin performance measures` – `end performance measures`. The entries of this block, each occupying one line, have the syntax from the standard output format

  `<name> <type>` or
  `<name>=<type>`

  where `<name>` is the name of the performance measure and `<type>` is its type. This information is extracted and provided by calling function `PerfMeasureOut()`. The return value of this function will be an array, which will be empty in case of an error. For each performance measure found in the performance measures block, an element is added to the return array. The element's key is the name of the performance measure, the element's value is the type in the form of its string encoding.

- `ParamsOut()`

  Each job's algorithm is run with a certain setting of its parameters. Modules, especially the last module of an algorithm, might print the parameter value pairs in a parameters block as described in the standard output format (see subsection 2.3.1 on page 24). If these parameter value pairs have the syntax from the standard output format

`<name> <value>` or
`<name>=<value>`

they can be extracted by calling function `ParamsOut()`. The parameters extracted will be stored in an array which then is returned by the function. For each parameter found in the parameters block, one element is added to the returned array. The element's key is the name of the parameter,its value is the scanned value of the parameter. If an error occurs, the returned array will be empty. Note that `ParamsOut();` is just an alias for `$this->ExtractParams($resulttext);`

- `$params`

  This data structure is an array holding all parameters and their values as known by the testbed. The element keys correspond to the parameter names, the element values to the values of the parameters, as they are known by the testbed. The number and names of these parameters might be different from the parameters as contained in the job result in the parameters block. The parameter names extracted and stored in variable `$params` are the parameter names as the testbed looks at it. These might differ from the flags used to call the algorithm on command line level and the names for the parameters the last module of an algorithms gives them. The names appearing in variable `$params` are the ones are constructed by the testbed starting with the names as defined in the module definition file of the modules of the job's algorithm. See subsection 3.3.6 on page 107 and subsection 4.2.3 on page 187 for details. Therefore, variable `$params` will always encompass parameters of modules other than the last module an algorithm, too. For example, the file name of a job's problem instance can be accessed with `$instance = $params["input"];`. Any other parameters can also be accessed with the corresponding name. Parameter name convention are described in the subsection about specifying configurations in subsection 3.3.6 on page 107 on page 110. Note that PHP function `array_keys(...)` lists all keys of an array, whereas function `array_key_exists(...)` returns, whether a key is used in an array or not.

- `Call()`

  Function `Call()` returns the command line call from the job result as defined in the call block enclosed by brackets `begin call` and `end call` from the standard output format. The return value is a string. It will be empty, if an error occurs or if the block or its contents are missing.

- `Solution()`

  This function returns the data extracted from solution blocks for the tries. It is an array of array, the outer array indexed by try number. The inner arrays store the data of each solution block in their elements. According to the standard

output format, reserved names `seed`, `solution` and the names of the performance measures of the command line interface definition of the last module of the job's algorithm are used to access the according values. Further name value pairs will be extracted as long as they are conform to the

`<name> <value>` or
`<name>=<value>`

format of the testbed standard output format. The command `Solution()` is an alias for `GetInfo('solution', $resulttext);`

Example:

```
$solution = Solution() //$solution = GetInfo('solution', $resulttext);
```

will yield the same result as the following assignment:

```
$solution = array (
    [1] => array ("seed" => 181,
                  "solution" => "{...}",
                  "best" => 2455, ...
                 ),
    [2] => array ("seed" => 182,
                  "solution" => "{...}",
                  "best" => 2449, ...),
        .
        .
        .
);
```

- `$resulttext`

  Variable `$resulttext` contains the complete text of the job result in the form of a string. This can string can be used to do additional scanning of the job result independent of the command provided by the testbed.

- `GetInfo($name,$text)`

  The contents of generic blocks in job results will not be extracted automatically. Instead, extraction of data in these blocks must be requested explicitely in a script. The contents of all generic blocks with name `<name>` is extracted from the job result by calling `GetInfo($name,$resultext)`. Recall that variable `$resulttext` contains the complete text of the job result in the form of a string. Command `GetInfo(...)` can be used with arbitrary strings as second argument. The result of this command will be an array of arrays. For each block found, the resulting array will have one element. The element's name is the number of the generic block, the elements value is an array of name value pairs representing the lines of the blocks.

Example:

Suppose, the following generic blocks are contained in the job result:

```
begin lpg 2
ffActions 97
ffLength 97
end lpg 2
begin lpg 1
lpgActions 181
lpgLength 181
end lpg 1
```

The result of executing `$info = GetInfo("lpg", $resulttext)` will be the same as the following assignment to variable `$info`:

```
$info = array (
    [1] => array ("lpgActions" => 181, "lpgLength" => 181),
    [2] => array ("ffActions" => 97, "ffLength" => 97)
);
```

- `compute( "<resultname>", "<statistic>", "<calculate on field>" )`

  It is possible to compute the minimum, maximum, average and other statistics over all values of specific fields of all lines that were stored to variable `$result`. Typically, these lines were added with command `addresult` while looping over all tries and lines accessing the field values found in the lines by variables `$row` and `$lastrow`. The statistics computed will be stored immediately in the final result table (`$retval`), bypassing the typical procedure of computing this table with commands `list(...)` and `listall()` as described next. Several `compute(...)` commands can be executed, but at the moment it is not possible to use both `compute(...)` and `list(...)` simultaneously. The following statistics are available to substitute for `<statistic>`. They are computed over all values available for the field `<calculate on field>` of all lines in array `$result`:

  - `max`/`maximum`: Maximum
  - `min`/`minimum`: Minimum
  - `avg`/`average`/`mean`: Mean/Average
  - `median`: Median
  - `quartile1`: First quartile
  - `quartile3`: Third quartile
  - `variance`: Variance
  - `stddev`: Standard deviation

Sometimes, the performance measure to compute the statistics for are to be input by the user and hence not known by the script in advance but stored in a variable. In such a case, it is not possible to simply use command `compute` with the variable as third argument, since the arguments of compute are taken literally: The variable name becomes the name of the performance measures the computation is based upon instead of the contents of the variable. In case the performance measure name is contained in a variable, the following commands can be used instead:

- `$retval["Minimum"] = $mathobj->min($result,$perfMeasures);`
- `$retval["1stQuartile"] = $mathobj->quartile1($result,$perfMeasures);`
- `$retval["Median"] = $mathobj->median($result,$perfMeasures);`
- `$retval["Mean"] = $mathobj->mean($result,$perfMeasures);`
- `$retval["3rdQuartile"] = $mathobj->quartile3($result,$perfMeasures);`
- `$retval["Maximum"] = $mathobj->maximum($result,$perfMeasures);`
- `$retval["Variance"] = $mathobj->variance($result,$perfMeasures);`
- `$retval["StdDeviation"] = $mathobj->stddev($result,$perfMeasures);`

Note again that the arguments of this command will be taken literally. That is, if the last argument is given by a variable, say `$arg`, holding string `Test`, only fields named `$arg` will be considered and not fields named `Test`! If this is not intended, the computation must be reprogrammed explicitly (see paragraph about table formats in this section on page 193 and the description of the next item).

- `list( "<fieldname 1>", ... , "<fieldname n>")`

  Arbitrary lines in the form of arrays with any number and label of elements can be added to variable `$result` which holds the present results of the data extraction effort. These lines were added manually by direct array element assignment or by using command `addresult(...)`. As described at the beginning of this subsection in the paragraph about the table formats of the data extraction process, the final result of the data extraction process on a job result must be organized along fields instead of lines as is the case with variable `$result`. For this reason, variable `$result` basically is transposed. The result of this operation is stored in variable `$retval` which contains the ultimate data extraction result of a job. The transformation is triggered by command `list(...)`. Since not all fields of any line ever added to the result is of interest in the final result, all fields not contained in the argument list of command `list(...)` will be discarded. Note that the arguments of this command will be taken literally. That is, if an argument is given by a variable, say `$arg`, holding string "`Test`", only fields named "`$arg`" will be considered and not fields named "`Test`"! If this is not intended, variable `$retval` has to be accessed directly by reprogramming command `list(...)` explicitly. Suppose

variable `$listp` is an array containing the field names not to be discarded during the transformation. Then, the following program fragment will do the transpose operation considering all fields listed in `$listp`

```
$lineNo = 0;
$foreach ($result as $data) {
    $lineNo++;
    foreach($listp as $param) {
        $name = trim($param);
        $retval[$name][$lineNo] = $data[$name];
    }
}
```

- `listall()`

  If no field is to be discarded when creating the final result of the data extraction process for a job, this command can be used. It is identical to command `list(...)`, except that it first finds all field names occurring in `$result` and next calls `list(...)` with all field names found. Note that columns automatically added to array `$result` with the help of `GetInfo(...)` and `addresult(...)` can only be transformed into the final output table by command `listall(...)`.

4.3.3 Examples

This paragraph presents some examples of how to write data extraction scripts. The examples are available as XML exports located in directory `DOC_DIR`*/scripts/extraction*. They are named *Usermanual-Example-1.X.xml* through *Usermanual-Example-4.X.xml*.

Example 1 This example demonstrates how to extract each last line of each try and how to finally return all values that were collected. The empty brackets `begineachrow{ , }endeachrow` must not be omitted since otherwise `$lastrow` will not be set correctly. The lines of the tries are supposed to comprise at least fields named `best` and `time`.

```
begineachtry{
    begineachrow{
    }endeachrow
    addresult( $lastrow );
}endeachtry
list("try", "time", "best");
```

If the last line is changed to

```
// Compute summary statistics
compute( "Minimum", "min", "best" );
```

```
compute( "Quartile-1", "quartile1", "best" );
compute( "Mean", "mean", "best" );
compute( "Median", "median", "best" );
compute( "Quartile-3", "quartile3", "best" );
compute( "Maximum", "max", "best" );
compute( "Variance", "variance", "best" );
compute( "Std.deviation", "stddev", "best" );
```

some summarizing statistics of field `best` are calculated and will be the only dataset added to `$retval` instead of the n (depending on the number of tries) datasets, which are generated by `list`.

Example 2 This example whose source code is given next shows how to stop processing the lines of each try, if a certain condition is met. This example assumes there as performance measure `best` recorded and each record has a field named `time` in some lines representing the cumulative runtime. The script is aimed to extract the first result that occurred after running for at least `n` second, with `n` requested interactively from the user.

```
<!--userinput-->
$userinput["stopTime"] = array(
    "description" => "Stop time:",
    "default" => "10.0"
);
<!--/userinput-->
$perfMeasures = "best";
begineachtry{
    $added = 0;
    begineachrow{
        if ( $row["time"] > $stopTime ) {
            // Stop time overstepped => add result
            addresult( $row );
            $added = 1;
            break;     // Leave row block.
        }
    }endeachrow
    if ( $added == 0 )  {
        // Criteria has not matched => Use last row,
        // e.g.~use best result that was found before
        // $stopTime seconds expired.
        addresult( $lastrow );
    }
}endeachtry
listall();
/*list( "try", "time", "best");*/
```

Example 3 The last example presented next shows how to calculate the solution quality in percent deviation from the optimum with respect to field `best`. This example is only usable for problem types which store the optimum in input files and which hence is accessible. For this example the optimum must be the last number in the first line. For other cases the regular expression must be adopted to match the optimum.

```
$pi = CreateObject("probleminstances.soprobleminstances");
$pidata = $pi->GetData($params["input"]);
if (preg_match('/^[^\n]*\s+(\d+)\n/i', $pidata["data"], $matches) )
$optimum = $matches[1];

begineachtry{
    begineachrow{
        // Avoid division by zero.
        if ( $row["best"] != 0) {
            $row["optimump"] = $optimum / $row["best"];
            $lastNonZeroRow = $row;
        };
    }endeachrow
    if ($lastNonZeroRow) {
        addresult( $lastNonZeroRow );
    } else {
        $lastrow["optimump"] = 0;
        addresult( $lastrow );
    }
}endeachtry

list("try", "time", "best", "optimump");
```

Assume an algorithm can retrieve the value for the optimum and outputs it in the job result in the form of a name value pair in the parameters block as is the case for the example algorithm called 'Dummy' that was created in subsection 3.2 on page 74 'Getting Started'. Let `optimumBest` be the name for the optimum for performance measure best. Then, the first line before the try- and row-blocks can be substituted with:

```
$parameters = ParamsOut();
$optimum = $parameters["optimumBest"];
```

Example 4 The purpose of the final example is to illustrate the usage of functions and predefined variable not covered in any example yet. The script will work fine with job results obtained with the dummy algorithm from section 3.2 on page 74.

```
// Increase time limit
set_time_limit(60);
```

```
$call = Call();
echo "<br>CLI call:<br>$call<br>";

// Output command line parameters
echo "<br>CLI parameters:<br>";
foreach($params as $paramName => $paramValue) {
    echo "Name: $paramName Value: $paramValue<br>";
}

// Output parameters as printed to the job result
echo "<br>Job result parameters:<br>";
$outputParams = ParamsOut();
foreach($outputParams as $paramName => $paramValue) {
    echo "Name: $paramName Value: $paramValue<br>";
}

// Output performance measure names and types as exported
// CLI definition of the modules.
//$perfMeasures = array("A","B","C");
//$perfMeasureTypes = array("TA","TB","TC");
//echo"<br><br>";
//print_r($perfMeasures);
//print_r($perfMeasureTypes);
echo "<br>CLI performance measures:<br>";
$size = count($perfMeasures);
for($i = 0; $i < $size; $i++) {
    echo "Name: $perfMeasures[$i] Value: $perfMeasureTypes[$i]<br>";
}

// Output performance measure names and types as printed in the
// job results.
echo "<br>Job result performance measures:<br>";
$outputPerfMeasures = PerfMeasuresOut();
foreach($outputPerfMeasures as $perfMeasuresName => $perfMeasuresValue) {
    echo "Name: $perfMeasuresName Value: $perfMeasuresValue<br>";
}

// Output solution as printed in the job results.
echo "<br>Solutions:<br>";
$solution = Solution();
//$solution = GetInfo('solution', $resulttext);
foreach($solution as $tryNo => $tryArray) {
    echo "Try: $tryNo<br>";
    foreach($tryArray as $solName => $solValue) {
```

```
        echo "Name: $solName Value: $solValue<br>";
    }
}

// Output contents of block Test as output by the testbed dummy module.
// job results.
echo "<br>Test block:<br>";
$testBlock = GetInfo('Test', $resulttext);
foreach($testBlock as $tryNo => $tryArray) {
    echo"Try: $tryNo<br>";
    foreach($tryArray as $blockName => $blockValue) {
        echo "Name: $blockName Value: $blockValue<br>";
    }
}
```

4.3.4 Further Information

To conclude this subsection about writing data extraction scripts, further information such as hints for troubleshooting and writing more elaborate scripts is given.

- All commands described in this subsection are mapped textually into a set of PHP-commands. The mapping of the commands can be viewed in file *class.boresultparser .inc.php* in directory `TESTBED_ROOT`*/statistics/inc/*. The functions for computing statistics over a performance measure as employed by macro `compute()` are contained in file *class.mathfuncs.inc.php* in directory `TESTBED_ROOT`*/common/inc/*.

- Do not comment out commands, i.e. macros such as `list`, `compute` with comments `//`, but use comments `/*` and `*/` instead. Comments `//` only comment out the line they are place in. Since commands are textually replaced, possibly ranging over several lines, only the first lines actually will be commented out, leading to a corrupted script. For this reason, never use commands in any commented region.

- If an element of an array such as `$result` or `$retval` has to be discarded, this can be done using function `unset(...)`, which takes as argument the variable holding the array. Suppose `<index>` is the key (either the appropriate string or an index number) for the element of array `$retval` that is to be discarded. The element can then be removed with

  `unset($retval[<index>])`

- The functions of PHP for printing to standard output such as `echo` or `print_r` can be used within an data extraction script, too. This output will be presented before the extraction script output is placed. It will be interpreted by the browser as HTML code. Newlines, for example, can be generated by printing string <br>.

Ordinary PHP newlines ('`\n`') do not work. If it is unclear what function extracting parts of the raw data of job results will return, just print the result with PHP function `print_r`.

- As well as any output to standard output in a data extraction scripts shows up on the page before the real data extracted is displayed, error messages occurring during the execution of a script, e.g. PHP error messages, will be displayed before the real data extracted is displayed, too.

- It is not mandatory to use function `addresult(...)` only for filling intermediate table `$result`, neither is it mandatory to use functions `compute(...)` or `list(...)` only to transform table `$result` to final table `$retval`. These tables can be accessed directly as shown in the discussion for function `list(...)`. However, the required table format, in particular the format of final table `$retval`, must not be disobeyed.

- Any additional output of data extraction scripts, for example, if PHP printing functions were used or if an PHP execution error occurred will be placed before the actual data extraction output. If the data extracted is to be viewed with option 'View Result in HTML' in submenu 'Data Extraction' (see 3.3.10 on page 125), this constitutes no problem. Export to CSV or to the other formats that result in storing the data extracted to disk , however, will not be possible, since the extra output is stored too, corrupting the CSV file format.

- If, after checking a data extraction script by pressing button 'Check Script', an empty page appears, an unknown function or command has been used. Any regular function of PHP and the functions described in this subsection are known functions. In this case, the 'Back' button of the browser can be used to get back to the script input page with the old values still filled in the input fields. The syntax check can, however, only reveal the existence of a syntax error, but neither type nor precise location. In order to localize the error, parts of extraction scripts can be commented out. Everything between `/*` and `*/` is ignored (even over newlines).

- Note that empty scripts are not allowed. A simple empty comment `//` will do, however. The error message coming up when trying to insert an empty script is `ERROR: ExecAppend: Fail to add null value in not null attribute script`.

- The comparison operator of PHP is `==` ! It is not `=` ! Using `=` will always evaluate to true.

- By means of the user input request facilities of data extraction scripts it is possible to equip the scripts with some generic functionality. Sometimes, however, it is cumbersome to repeatedly type in the same information again and again, especially if the script is rerun several times with the same settings. This can be allayed by

changing the script such that the repeatedly required values for the user inputs become the default values of the user input requests. Another possibility is use the copy and edit mechanism for objects in the testbed. That way, a generic script, perhaps without any user input request at all, is written which as is can not be run properly. Instead, it must be copied and the necessary adjustments are written in the script directly in the form of variable assignments to script internal reserved variables. This makes sense, if the script is to exhibit a larger degree of generic support. Once such a generic script was adjusted for a specific experiment, it is stored together with the experiment data and specification and can be reused without any changes later to reproduce the statistical evaluation. This approach was taken for some generic analysis scripts (see next subsection). Both approaches, using changing default values of the user input requests and changing some settings in the script are essentially the same, since they both involve changing the script. In order not to loose settings, the script has to be copied. If the settings are more comprehensive, however, possibly involving more complicated data structures, the second approach seems to be more straightforward.

- When developing data extraction scripts as well when developing analysis scripts, changes in the default settings of user input requests will only will take effect, if temporarily another script is chosen in submenus 'Data Extraction' or 'Data Analysis', respectively.

- PHP restricts the runtime of processes. Sometimes, however, a data extraction script needs more time to compute the results, either because the computation is very complicated or because the set of job results processed is extensive. In order to prolong the runtime allowed, the following PHP command can be used as the fist command in a data extraction script:

  ```
  set_time_limit(x);
  ```

  Argument `x` to this command must be an integer indicating the runtime allowed in seconds. The maximum runtime for PHP processes can be changed permanently in file */etc/php.ini* under item `max_execution_time`.

- If the data extraction does not yield a result for command `compute( "<resultname>", "<statistic>", "<calculate on field>")` there are some few typical reasons:

  - The statistics `<statistic>` might be misspelled in the compute statement.
  - Field `<field>` does not exist.
  - The textual replacement of the arguments for `compute() <resultname>`, `<statistic>`, or `<calculate on field>` is not as intended. Compare to the discussion of the `compute()` command.

- It is a good idea to enclose string always in double quotes '"'. Single quotes '' behave differently and in general not straightforward, for example, if variables are to be expanded in the enclosed string.

- The testbed requires the PHP options *magic_quotes_gpc* to be set to *off* in file */etc/php.ini* (compare with subsection 3.1.3 on page 61). if the magic quotes are *on*, some strange behavior can exhibited by the testbed. In particular, when checking or adding extraction or analysis scripts, the typical comments `//` and `#` might not work anymore. Additionally, any quote `"` will be replaced by `\"`.

- If an extraction script employed exhibits as parse error (which would have been detected, if the script had been checked with button 'Check Script'), the following error message will show up instead of the extraction result:
```
Parse error: parse error in
 /usr/local/httpd/htdocs/testbed/statistics/inc/
class.boresultparser.inc.php(220) : eval()'d code on line 169
```

- Each time an extraction script is employed, it is run on an empty dummy job result first, i.e. variable `$resulttext` contains no text. In order to bypass any processing during the dummy run, enclose the script in `if($resulttext != "") { ... }`.

- Data extraction and analysis script descriptions are best viewed with the text area comprising at least 60 columns. See subsection 3.3.12 on page 132 for information about changing this setting.

- See also the further information of section 4.4 on the following page and the troubleshooting section 4.6 on page 217.

4.4 Writing Analysis Scripts

The final interface that were identified in section 2.2 on page 11 concerning the requirements of a testbed that has not been discussed completely yet is the interface between the extraction of specific data from the output of the algorithms run and the subsequent statistical evaluation. This subsection explains how to write scripts in the R language [60] for conducting the statistical evaluation. Search filters select specific sets of job results from the set of all job results contained in the testbed. Data extraction scripts then extract the relevant data from the selected job results. This data is intended to be used as input for further statistical analysis. The data is transformed into a table similar to tables in relational databases. This table then is input into statistics package to conduct a statistical evaluation. In case of the testbed, the R statistics package [60] is used. There are two possibilities to convey any data extracted to R:

1. Storage of the data extracted as a CSV file to the file system and external operation of R with this exported data.
2. Usage of an analysis script from within the testbed which addresses the R package directly and which imports the data to R automatically. R functions can be called externally from PHP. That way, the testbed can executed analysis scripts.

Analysis scripts almost exclusively comprise R language constructs. For example, to comment out a single line, use `#`. Only two add ons to the R programming language were designed for this testbed. The add ons are responsible for provision of user input and enable to load the proper data into R if addressed directly by the testbed. These two add on are described next. Information and documentation about the R-language that is employed in the analysis scripts can be found on the R home page [60]. The documentation about S/S-PLUS can also be used since R is a free clone of S-PLUS. Various books about R and the compatible S/S-PLUS package are available [19, 20, 21, 22, 23, 24]. Reading the R help mailing list can also be a good start for beginners, because a lot of code snippets are posted there.

User input is requested the same ways as is done for data extraction scripts (see previous section). In fact, the example given when discussing user input request for data extraction scripts can be used one-to-one in an analysis script, too. The only difference is a different incorporation of the user input. While data extraction scripts are PHP programs themselves, an analysis script is not. Therefore, with respect to the example given for the user input request in the last subsection, variables `$Test1` and `$Test2` can not be used in the R script. Instead, pseudo variables `{Test1}` and `{Test2}` can be used. These are not variables that will be really recognized by R as variables. Instead their contents is substituted *textually* before execution of the script wherever they are positioned. For example, if the user had input `c(1,2)` for user input request `Test1`, lines

```
test <- {Test1};
cat("\nTest:",test);
```

would have worked out producing output

```
Test: 1 2
```

since `c(1,2)` is valid R construct to be used as right hand side of an assignment (`<-` is the assignment operator in R, `c(...)` is a construct to build arrays in R).

On the other hand, if the user had input `Test`, R would have output an error message like this:

```
Error: Object "Test" not found
Execution halted
```

Usage of a pseudo variable `{XYZ}` in a string is harmless.

Pseudo variable `{inputfilename}`, is predefined and specifies the name including the path to the input file that contains the data extracted or rather that data that is to be processed. In fact, the testbed exports the data supposed to be imported into R when addressing R directly from within the testbed to a temporary file, too. It then has to be loaded in the analysis scripts as if it has been exported by the user. Since in this case the name and location of the temporary file can not be known in advance, pseudo variable `{inputfilename}` can be used to access the temporary file. This variable should always be used for reading the input when using R scripts from within the testbed, because depending on the system configuration the input file has different locations. This pseudo variable is set by the testbed. The usage of pseudo variable `{inputfilename}` is depicted with the following code fragment:

```
# Read data
inputdata <- read.csv("{inputfilename}", sep=",");
```

If addressing R by the testbed directly by means of an analysis script in submenus 'Data Extraction' and 'Analyze Data' (see subsections 3.3.10 on page 125 and 2.3.4 on page 34 on page 34, , respectively), the format of the temporary file containing the data extracted will always be that of a comma separated CSV file.

Examples of analysis scripts can be found in directory `DOC_DIR/examples/scripts`. In directory `DOC_DIR/scripts/` several useful generic scripts for plotting curves and box plots, and for doing statistical testing can be found. Since it is straightforward to share analysis scripts via the testbed it is expected that an increasing number of useful scripts, more or less tuned for use with the testbed will be developed and shared in later versions of the testbed. Note that in principle, any analysis script can be used without major change standalone, too. The only changes that have to be done concern the two add on to the R programming language by the testbed.

In what follows in this subsection, some hints and guidelines for using analysis scripts most efficiently are given.

4.4.1 Further information

- Some generic scripts for plotting of curves and box plots and for doing pairwise parametric and non-parametric statistical testing can be found in directory `DOC_DIR`*/scripts/analysis*. They all end with *.R.xml*.

- HTML- and PHP language constructs other than the brackets used for encapsulating user input requests can have spurious result. For example, using a wrong user input format in brackets `<!--userinput-->` and `<!--/userinput-->` that request user input can have the result that submenu 'Data Extraction' can not be displayed any more.

- Changes in user input requests such as changing a default value will only take effect after the R changed script has been reloaded by first selecting a different analysis script and subsequently changing back to the changed one (compare with writing data extraction scripts discussed in the last subsection).

- If an analysis script contains syntax errors, the script will not be executed by R. Instead, an error message like the following two lines will be returned:

  ```
  Error: syntax error
  Execution halted
  ```

- Unfortunately, it is not possible for the testbed to retrieve the number of the line where a syntax error occurred, although R provides this information when used standalone. Each time an analysis script is executed from within the testbed, the data to be processed has to be extracted, conveyed to R via a temporary file and loaded into R, before the actual analysis script can be executed. This is very cumbersome compared to a standalone usage of R. In this case, the data to be processed has to be extracted, exported and loaded into R only once. For these reasons, it is preferable to develop analysis scripts for the testbed using directly. R can read in script files with command `source` and will execute them. If a syntax error occurs, R will give the number of the problematic line. Furthermore, using R directly gives access to the R internal help facility. All R functions are explained comprehensively in man page style. Recapitulating, it is recommended to develop analysis scripts using R directly.

- The analysis scripts are valid analysis scripts for standalone use of R, too. An analysis script for standalone use is constructed by copying the script from the editing page for analysis scripts to another file and setting variable `testbedScript` to true with `testbedScript <- T`. The script can then be applied using the `source` command of the R language. Such a script simple has to be copied from the testbed using the clipboard or by removing XML tags from an XML export. The command the looks like:

```
source("/path/to/script/R-Script.R")
```

Before, the data to analyze should be loaded into R. This is done by first exporting it to csv format and next starting R on the command line interface with `R` and the loading the data into R with

```
inputdata <- read.csv("/path/to/data/data.csv", sep=",");
```

Next, the script can be executed.

- As was discussed in the last subsection about writing data extraction script, interactive user input requests make only sense for a small amount of user input required. If more comprehensive user input is required, possibly involving complex data structures, the copy and edit approach to analysis script parameterization is more suitable. This approach requires to copy a generic script, changes some settings in the form of changing data structure at the beginning of the script which then will control the process of the script. This approach was used for the generic scripts that are provided in directory `DOC_DIR`*/scripts/analysis/*. Even if it seems cumbersome, too, it enables a purely declarative instantiation of analysis scripts and provide very powerful generic scripts.
- If any analysis script will finally produce some output file, e.g. plots, the names of theses output files should not contain special shell character or command such as the piping character '`|`', because these will be interpreted by the shell when R is trying to store such files. An error will occur and the file is not written to disk.
- Empty analysis scripts are not allowed.
- See also the further information of section 4.3 on page 192 and the troubleshooting section 4.6 on page 217.

4.5 Web Interface for the Database

A web front end is available for PostgreSQL databases. It is called phpPgAdmin [57]. To install this front end to the local host, one has to carry out the following steps:

- Download the compressed files from *http://phppgadmin.sourceforge.net/*.
- Extract the files into the directory containing the local web pages, which will probably be `/usr/local/httpd/htdocs/` in a SuSE installation.
- Create a file named `.htaccess` in the directory of the web front end which typically is `phpPgAdmin`. File `.htaccess` contains only one single line:
  `magic_quotes_gpc = On`

- Copy the configuration file for the PostgreSQL web front, `config.inc.php-dist` to a file named `config.inc.php` and change the following lines as stated here. In this example, the database employed, the user and the password for this user are `testbed`. In general, the settings made in the personal testbed configuration file *./testbed.conf.php* have to entered here instead. The configuration file is well documented. See there to change it to suit individual needs. In the following, an example is shown:

```
$cfgDefaultDB   = "testbed";
$cfgServers[1]['local']         = true;
$cfgServers[1]['host']          = 'localhost';
$cfgServers[1]['port']          = '5432';
$cfgServers[1]['adv_auth']      = true;

// if you are not using adv_auth, enter the username
// to connect all the time
$cfgServers[1]['user']          = 'testbed';
// if you are not using adv_auth and a password
// is required enter a password
$cfgServers[1]['password']      = 'testbed';
// if set to a db-name, only this db is accessible
$cfgServers[1]['only_db']       = 'testbed';
```

- Access the web front end with a web browser (typically reached via URL or file `http://localhost/phpPgAdmin/index.php`) and enter `testbed` as both username and password.

4.6 Troubleshooting and Hints

This section is a loose collection of possible causes for a testbed malfunction together with strategies to cope with the malfunction or how to find and remedy the cause. Additionally, some guideline for using the testbed efficiently and for getting past some limitation of the testbed are given:

- If the testbed exhibits strange behavior, it is always a good idea to first empty the cache. Additionally, reentering the testbed's home URL instead of trying the 'Back' button of the web browser might solve any behavioral problems.
- Error messages from the testbed always begin with

  `Fatal error:`

  for example

  `Fatal error: Call to undefined function: getneededuservars() in /usr/local/httpd/htdocs/testbed/statistics/inc/class.uirscripts.inc.php on line 359`

  These will show up on a page instead of the contents originally expected.
- If the directory as specified in files `TESTBED_ROOT/`*config.php* or *~/.testbed.conf.php* that is used to store the result files jobs produce before these are stored back into the database can not be created, for example because the access rights are not set properly, the following error will occur when starting the job server:

  ```
  <br />
  <b>Warning</b>:  mkdir() failed (Permission denied) in
  <b>/usr/local/httpd/htdocs/testbed/jobs/inc/class.bojobs.inc.php</b>
  on line <b>56</b><br />

  No Jobs in queue, waiting ...
  ```

  In such a case, it should be checked, whether the directory specified in the two file mentioned is correct and if this is the case, whether the access rights are set properly, for example with `chmod` orwx /tmp/testbed-user+
- Parameters of type FILENAME must contain path information when given through configurations of the testbed (see subsection 3.3.7 on page 110 and paragraph 2.3.1 on page 15), otherwise the files will not be found since it they are looked for in a temporary directory created by the testbed on demand. The testbed server then will mixup the given filename for the name of the problem instance file and might output the following error message: `Could not retrieve problem instance 'XYZ'!` where `XYZ` is the filename that has no proper path information. It is best to use absolute path information.

- If job and hence experiment statuses seems to be wrong, it might be necessary to reset the testbed. This is done via the CLI with command

  `testbed reset`

  As a result of this command all jobs with status 'running' are set to status 'FAILED' and can thus be restarted. Everything else does not change. In particular, the job execution queue remains unchanged. The reason to reset the testbed could be that a job obviously is not running anymore, because its process was killed. A job serverwill not recognize this and hence can not update the job's status to 'FAILED'. Since a job can only be restarted, if its status is 'finished' or 'FAILED', a 'lost' job can not be restarted. Information about setting the maximum runtime for jobs can be found in section 3.4 on page 136 on page 140. A complete discussion of job and experiment statuses can be found in subsections 3.3.8 and 3.3.9 on pages 116 and 121, respectively. Job servers as discussed in subsection 3.4.4 on page 140 also.

- If a job fails to execute, it could be that the output file of a module is not properly written to the place and name as set by the testbed with the help of parameter `--output`. The output file then is not written at all or to a wrong place and/or name. In any case, the testbed is not able to find the final output file and can accordingly not store the job result back into the database or convey it to the next module of an algorithm. This kind of error is indicated by the following messages on standard output by a job server (in the two examples following, each job's algorithm consists of a sequence of two jobs, for each the second not working properly):

```
/============================================================\
===================== Working on Job #998 ====================
\============================================================/

   Module: DummyOK
--------------------
Executing: '~/Testbed/bin/i386/linux/DummyOK/DummyOK
            --finallyFail 0 --finallyWait 0
            --input "/tmp/user-testbed/jobs/998/100.Dummy.dat"
            --output "/tmp/user-testbed/jobs/998/1/100.Dummy.dat"'

Progress: .

Execution of module succeeded!

   Module: DummyWrongOutput
----------------------------
```

```
Executing: '~/Testbed/bin/i386/linux/DummyWrongOutput/DummyWrongOutput
            --finallyWait 0 --finallyFail 0
            --input "/tmp/user-testbed/jobs/998/1/100.Dummy.dat"
            --output "/tmp/user-testbed/jobs/998/output.dat"'

Progress: .

Execution of module succeeded!

<br />
<b>Warning</b>:  fopen("/tmp/user-testbed/jobs/998/output.dat", "r") -
No such file or directory in
<b>/usr/local/httpd/htdocs/testbed/jobs/inc/class.bojobs.inc.php</b>
on line <b>203</b><br />

Job error:
        Could not open final result output file
        '/tmp/user-testbed/jobs/998/output.dat' of Job.

/================================================================\
=====================  Job #998 failed!  =======================
\================================================================/

No Jobs in queue, waiting ...
```

It could also be that a module does not handle the input file set with flag `--input` properly. in such a case, the following output to standard output would be produced by a job server.

```
/================================================================\
===================== Working on Job #999 ======================
\================================================================/

   Module: DummyOK
----------------------
Executing: '~/Testbed/bin/i386/linux/DummyOK/DummyOK
            --finallyFail 0 --finallyWait 0
            --input "/tmp/user-testbed/jobs/999/100.Dummy.dat"
            --output "/tmp/user-testbed/jobs/999/1/100.Dummy.dat"'

Progress: .

Execution of module succeeded!
```

```
   Module: DummyWrongInput
-----------------------------
Executing: '~/Testbed/bin/i386/linux/DummyWrongInput/DummyWrongInput
            --finallyWait 0 --finallyFail 0
            --input "/tmp/user-testbed/jobs/999/1/100.Dummy.dat"
            --output "/tmp/user-testbed/jobs/999/output.dat"'

Progress: .

        Couldn't open input file:
                          /tmp/user-testbed/jobs/999/1/100.Dummy.dat

Execution of module failed!

Errors:
        Command '~/Testbed/bin/i386/linux/Dummy/Dummy
                 --finallyWait 0 --finallyFail 0
                 --input "/tmp/user-testbed/jobs/999/1/100.Dummy.dat"
                 --output "/tmp/user-testbed/jobs/999/output.dat"'
        exited with return code != 0.

/============================================================\
=====================  Job #999 failed!  =====================
\============================================================/

No Jobs in queue, waiting ...
```

The job's status will be 'FAILED' afterwards and viewing the job's output will only show the following error messages:

```
Could not open final result output file
       '/tmp/klausvpp-testbed/jobs/999/output.dat' of Job.
```

and

```
Command '~/Testbed/bin/i386/linux/DummyWrongInput/DummyWrongInput
         --finallyWait 0 --finallyFail 0
         --input "/tmp/user-testbed/jobs/999/1/100.Dummy.dat"
         --output "/tmp/user-testbed/jobs/999/output.dat"'
exited with  return code != 0.,
```

respectively.

In either case, either the module definition file or the executable itself have to be checked, whether the output is really stored at the place that was requested by the value for parameter `--output` or whether the input file handling works correctly.

- If the testbed does not respond anymore, a possible cause might be that the database server or the apache web server is not running properly or at all anymore (compare to next item for error message that will appear in such a case most likely). In this case, the user has to restart the database or the apache web server or both manually. To do this, the user has to log in as super user and issue commands (the commands are listed for SuSE and Debian Linux systems)

  Debian `invoke-rc.d postgresql restart`
  `invoke-rc.d apache restart`

  SuSE `rcpostgresql restart`
  `rcapache restart`

  Sometimes, even this does not restart the database or web server properly. Then, the servers have to be shut down and restarted manually with the following commands:

  Debian `invoke-rc.d postgresql stop`
  `invoke-rc.d postgresql start`
  `invoke-rc.d apache stop`
  `invoke-rc.d apache start`

  SuSE `rcpostgressql stop`
  `rcpostgresql start`
  `rcapache start`
  `rcapache stop`

  The status of the database and web server on the local machine can be checked with commands

  SuSE `rcpostgressql status`
  `rcapache status`

  If after restart the status of one of the servers is still unused or not running (they both must be running), they most likely have to be restarted by hand as described just now.

  After a restart of either the database or web server, it is recommended to shut down all job servers with 'Ctrl-C', reset the database with command

  `testbed reset`

  and start any job server anew.

This prevents that any jobs that were running when the testbed was reseted keep on running without the possibility that its results can be stored back into the database. Additionally, any jobs not started upon server restart might be obsolete. Finally, the job server might not be able to store the results of jobs it runs back into the testbed database, so their runs will be futile anyway.

- Note that when changing file `TESTBED_ROOT`*/config.php*, it should be changed directly. It might not work to change a copy and overwrite the old version. This might yield the following error:

  ```
  Database error: Link-ID == false, connect failed
  PostgreSQL Error: 0 ()
  File: /usr/local/httpd/htdocs/testbed/common/inc/class.db.inc.php
  Line: 127
  Session halted.
  Fatal error: Call to a member function on a non-object in
  /usr/local/httpd/htdocs/testbed/common/inc/class.db.inc.php
  on line 1168
  ```

  In order to avoid this error, file `TESTBED_ROOT`*/config.php* should only be changed directly, e.g. when changing the search mask (compare to subsection 5.3.2 on page 270). Of course, to do so, one has to have super user rights.

  This error will also appear, if the database is not running properly or not running at all.

  In general, such a database error will occur, if any part of the testbed can not connect to the database it is supposed to. This can concern the web interface as well as the command line tool.

- If error message

  ```
  PHP Fatal error:  Unable to start session mm module in Unknown on line
  0
  ```

  occurs after starting a job server, this indicates that the current version of PHP used is buggy. This is the case for the PHP shipped with the SuSE Linux 8.0 distribution. In order to remedy this bug, the PHP version has to be repaired. A suitable rpm file can be found via the home page of the testbed [62].

- The XML import and export facilities are quite powerful. Additionally, being able for the user to read XML export enables direct manipulation of exported objects of the testbed. This, for example, is useful for quickly editing large objects such as large scripts with an editor of the user choice instead of using the limited editing facilities of the testbed.

Copy & Edit actions can be done by exporting an object to XML, renaming the object exported in the appropriate field in the XML file and re-importing it into the testbed again.

Be aware of the subtleties involved when name clashes occur. If an object is to be exported, only those objects it depends on are possibly exported, too. During re-import, those additionally exported objects the originally exported object depends on are imported, if and only if no other object with the same type and name is already contained in the testbed, no matter whether these are in fact identical or not. The same problem arises when only links, i.e. names to objects were exported: The re-imported object might refer to wrong objects as being dependent on. This might have strange errors and behavior as a consequence. Compare to subsection 3.4.3 on page 139.

- A common problem when viewing submenus is that although objects of the submenu's type are present in the testbed, no one is shown, even if no filter was applied. Another common problem is that obviously a wrong set of entries is displayed. Changing the filter might only take effect, if temporarily another filter is selected. The reason for this behavior is that the testbed still uses an old filter and has to be ordered to change the filter. Perhaps, still an old current search filter is used.

- If the interface of a module or rather the interface of the module binary executable remains the same after the code of the executable has been changed, e.g. if bugs have been fixed, it can exchanged any time by just replacing the executable in the appropriate directory without having to generate and/or edit the module definition file anew. A Module definition file's internal mode of operation can be changed as well without having to re-registering it to the testbed as long as the interface remains the same. If the interface changed, the corresponding module has to be deleted first, then registered anew. Note that deletion of a module will delete all dependent objects, too.

- The status of the errors or problems that have occurred last can be accessed via the web front end of the testbed by changing the end of the URL used from `index.php` to `check.php`. Additionally, `check.php` gives information about some settings of the testbed, for example about the maximum memory limit for PHP processes The status pages are linked in the main menu of the testbed through submenu 'Testbed Status', too. See subsection 3.3.13 on page 133 for more details.

- If the memory the testbed is allowed to use is not enough, for example when executing a larger data extraction script, the limit can be changed in file */etc/php.ini* for a SuSE installation and in file */etc/php3/apache/php.ini* or */etc/php4/apache/php.ini* for a Debian installation under point `memory_limit`. Afterwards, the apache web server has to be restarted (see subsection 3.1.2 on page 54). Note that this memory

limit is valid for *each* PHP process started by the testbed on a machine. If a lot of such processes are started, which can easily be the case, for example in a multi user operation, the overall memory consumption might quickly become enormous (compare with subsection 3.1.4 on page 69).

- A good idea, especially when using the search filter generation tool, is to open one or more additional windows displaying different aspects of the testbed. This increases the overview over the testbed and can be used for easy and quick copy and edit operation, for example to fill input fields in the search filter generation mask. Note, however, that navigation through the browser's 'Back' button might be corrupted.

- If a newly created configuration does not show up anywhere in the testbed, the simply cause might be that it was not really created. On the last page of the configuration creations sequence, after the parameters were submitted, the actual creation must be confirmed explicitely by pressing button 'Create Configuration'. Leaving this page by clicking any other button or link of the main menu will cancel the creation!

- If using conditions when configuring an algorithm (see subsection 3.3.7 on page 110), the names of the parameters name convention (see subsection 3.3.6 on page 110) must be heeded exactly. If a parameter is misspelled in a text input field, the parameter settings of this text input field will be discarded resulting in less fixed parameter settings that expected. To recall, parameter names are constructed according to scheme `<Modulename>_#_<Parameterame>` with `<Modulename>` representing the name of the module as entered when integrating the module into the testbed, `#` representing the position of the module in the sequence of modules of the algorithm that is configured, and `<Parameterame>` being the long flag of the parameter as used and exported by the module.

- When extracting data in submenu 'Data Extraction' (see subsection 3.3.10 on page 125) and using the button 'Calculate Fields', the testbed sometimes gets confused with the column names and presents wrong names either in the selection list or in the final output. When this happens, empty the cache and reload the submenu again, however, not using the 'Back' button of the web browser.

- Some useful generic extraction and analysis scripts have been exported to XML and are located in directory `DOC_DIR`*/scripts*. Data extraction scripts are always named **.X.xml* while analysis scripts are named **.R.xml*. Example and other generic data extraction and analysis scripts illustrating some aspects of writing scripts, e.g. user input, are located in `DOC_DIR`*/scripts/analysis* and `DOC_DIR`*/scripts/extraction*, respectively. To import all scripts available in these two directories, import files *Standard-All.R.xml* and *Standard-All.R.xml*, respectively, since these contain all other scripts in multi export format.

- The functions of the testbed dummy written in C can be reused by means of copy & paste. Additionally, in directory `DOC_DIR`*examples/modules*, a compressed `tar` file (`DOC_DIR`*/examples/modules/Interfaces-Tools.tgz*) contains classes written in C++ that implement basic functionality with respect to parsing the command line parameters of a program call and outputting results in proper standard output format. The main classes for parsing parameters are named `Parameter` and `ProgramParameters` (files *Parameter.h*, *Parameter.cc*, *ProgramParameters.h*, and *ProgramParameters.cc*). They implement a convenient specification and parsing method for the command line interface of programs according to the command line definition format. These can be reused, too. Class `StandardOutputFormat` in the compressed `tar` file `DOC_DIR`*/examples/modules/Interfaces-Tools.tgz* (files *StandardOutputFormat.c* and *StandardOutputFormat.h* implements a convenient methods to output results in proper format according to the standard output format of the testbed (see paragraph 2.3.1 on page 24 of subsection 2.3.1 on page 14). File *Interfaces-Tools-Example.cc* implements a demonstration of how to use the interface tools. It can be compiled using command `make`. All other files in the compressed `tar` file are auxiliary classes or files such as *PerformanceMeasure.h* and *PerformanceMeasure.cc* implementing a class to represent performance measures, *RandomNumberGenerator.h* and *RandomNumberGenerator.cc* implementing a random number generator, and *Timer.h* and *Timer.cc* implementing timing functionality. All files are documented using the format of the Doxygen documentation system [37].

- In case of a Debian-Linux system, the following line must not occur twice in file */etc/php4/cgi/php.ini*:

  Debian `extension=pgsql.so`

  Note that this is sometimes done automatically upon installation of the PHP or PostgreSQL modules. If this line is doubly in the configuration file, the following error can occur:

```
PHP Warning: Function registration failed -
                     duplicate name - pg_connect in Unknown on line 0
    ...
PHP Warning: Function registration failed -
           duplicate name - pg_setclientencoding in Unknown on line 0
PHP Warning: pgsql: Unable to register functions, unable to load in
                                                    Unknown on line 0
<b>Database error:</b> Link-ID == false, connect failed<br>
<b>PostgreSQL Error</b>: 0 ()<br>
<br><b>File:</b> /var/www/testbed/common/inc/class.db.inc.php<br>
<b>Line:</b> 131<p><b>Session halted.</b>
```

- Environment variables `OSTYPE` or `HOSTTYPE` are needed by the command line version of the testbed. If they are not set properly, problems will arise since the job server can not find the platform and operating specific binaries (compare to subsection 2.5.4 on page 47) In order to set them properly the following code fragment with respect to the bash shell can be inserted into file */etc/profile*:

```
test -z "\$HOST" \&\& HOST=‘hostname -s 2> /dev/null‘
test -z "\$CPU"  \&\&  CPU=‘uname -m 2> /dev/null‘
test -z "\$HOSTNAME" \&\& HOSTNAME=‘hostname 2> /dev/null‘
test -z "\$LOGNAME"  \&\& LOGNAME=\$USER
case "\$CPU" in
    i?g86) HOSTTYPE=i386   ;;
    *)    HOSTTYPE=\${CPU} ;;
esac
  OSTYPE=linux
# Do NOT export UID, EUID, USER, MAIL, and LOGNAME
export HOST CPU HOSTNAME HOSTTYPE OSTYPE
```

  Alternatively, these they can be set by hand using a bash shell:

```
export OSTYPE=linux
export HOSTTYPE=i386
```

- In order to make the command line version of the testbed start and run more smoothly, the following link can be created:

  `ln -s php4 php in usr/bin (/usr/local/bin)`

- The testbed was tested extensively with Mozilla Version 1.1 and 1.3. The behavior of the testbed might change from browser to browser. Untypical errors might occur on some browsers.

- See also the further troubleshooting and hint information of sections 4.4 and 4.3 in subsections 4.4.1 and 4.3.4 on pages 4.4.1 and 4.3.4, respectively and the information given in subsection 3.1.3 on page 61.

5 Architecture

This chapter is intended for developers who want to extend the testbed and user who need deeper insight into the testbed functioning, for example in order to formulate more complicated search queries or to modify the search mask in submenu 'Search Filters'. First, the database structure of the testbed is explained. Next, the overall structure and the source code of the testbed's framework is described. Among the information given are discussions about important variables and classes as used in the implementation.

5.1 Database Structure

Any variable data of the testbed is stored and organized by a relational database based on PostgreSQL. Other database management systems may also be used with little further effort. In order to be able to extend the testbed as well as to generate more complex queries when searching for specific sets of objects, it is vital to know the database structure, i.e. its tables and relations, in detail. This section provides this information.

The main structure of the testbed database is shown in figure 5.1 on page 229. Each box in figure 5.1 represents a table in the database. The figure is based on an entity relationship (ER) diagram as described in [3]. Each *primary key* of a table is marked with a '`-`', a *foreign key* is marked with a '`#`' (possibly only visible as '`=`'). A table using a foreign key can be recognized by following the lines in the figure. Foreign keys have the same name as the primary keys from the table the foreign key refers to. A line representing a foreign key primary key relationship starts with a rhombus in the table containing the foreign key and ends with an arrow on the table with the primary key.

Figure 5.1 only contains the most important tables of the testbed. There are additional tables for statistics and category data types. Tables representing static categories for example have a relation to experiments, problem instances or configurations, but they are not shown, since the figure would become too complex and clouded. The tables for the static categories are called `<object-type>categories` and consist of two foreign keys. The first foreign key is associated with the category while the second foreign key is associated with the identifier of the objects they group together like the name of an experiment or problem instance. The statistics and category tables are not included

in figure 5.1 on the next page, because they do not have major relations to the tables storing the testbed's main object types.

Note: It may also be possible to use one table to store all the information about objects grouped in categories in one table. However this method would be slower and the cleanup of the references could not be done automatically by the database. On the other hand the benefit of this solution would be that the testbed is easier to extend, because additional object types can also use the existing table instead of requiring their own tables for storing static categories.

5.1.1 Generation of Search Queries

Figure 5.1 on the facing page can be used to develop more complex SQL statements implementing search filters than can be generated automatically with the help of the search mask in submenu 'Search Filters' of the testbed (see subsection 3.5.1 on page 146). The development of SQL statements for relating attribute value restrictions in arbitrary ways other than pure logical AND relations (see subsection 29 on page 150) can be undertaken as follows: First, all object types affected by the sought-after target set have to be connected. Any type of object that has an attribute contributing to the target set specification is concerned. Starting at one type and following the lines in figure 5.1 on the next page the user has to try to connect all types concerned. The object types or rather tables thus connected are combined by joins in the SQL statement using command `JOIN`. Note that joins of tables can only be used in an SQL select command (`SELECT`). Joins in SQL are typically denoted by

`<table1> JOIN <table2> USING ( <key in table1>, <key in table2> ).`

That way all types of objects are included into the statement and thus their attribute value restrictions can be combined. Beforehand, however, the table joins have to be constraint further, since with the optional `USING` command, a Cartesian product will be build. Using the `USING` command with keys `<key in table1>` and `<key in table2>` will filter out all tuples build by the Cartesian product that do not agree in attribute `key` of the object types represented by tables `table1` and `table2`. Note that both tables must feature an attribute `key`. This attribute typically is a primary key in one table and a foreign key in the other table. More about joins and foreign keys can be read in [26, 35].

After having related all object types concerned this way, the individual attributes of the object types can be assigned value restriction and subsequently these individual restrictions can be combined to form one final constraint. This final stage of query generation is specified with the SQL construct `WHERE` which works as a final filter on the tuples resulting from the join operation. Attribute value restrictions can be defined by `type.attribute = 'xyz'` meaning that the attribute `attribute` of object type `type` has to have value `xyz` which can be a number, a string , a regular expression or the

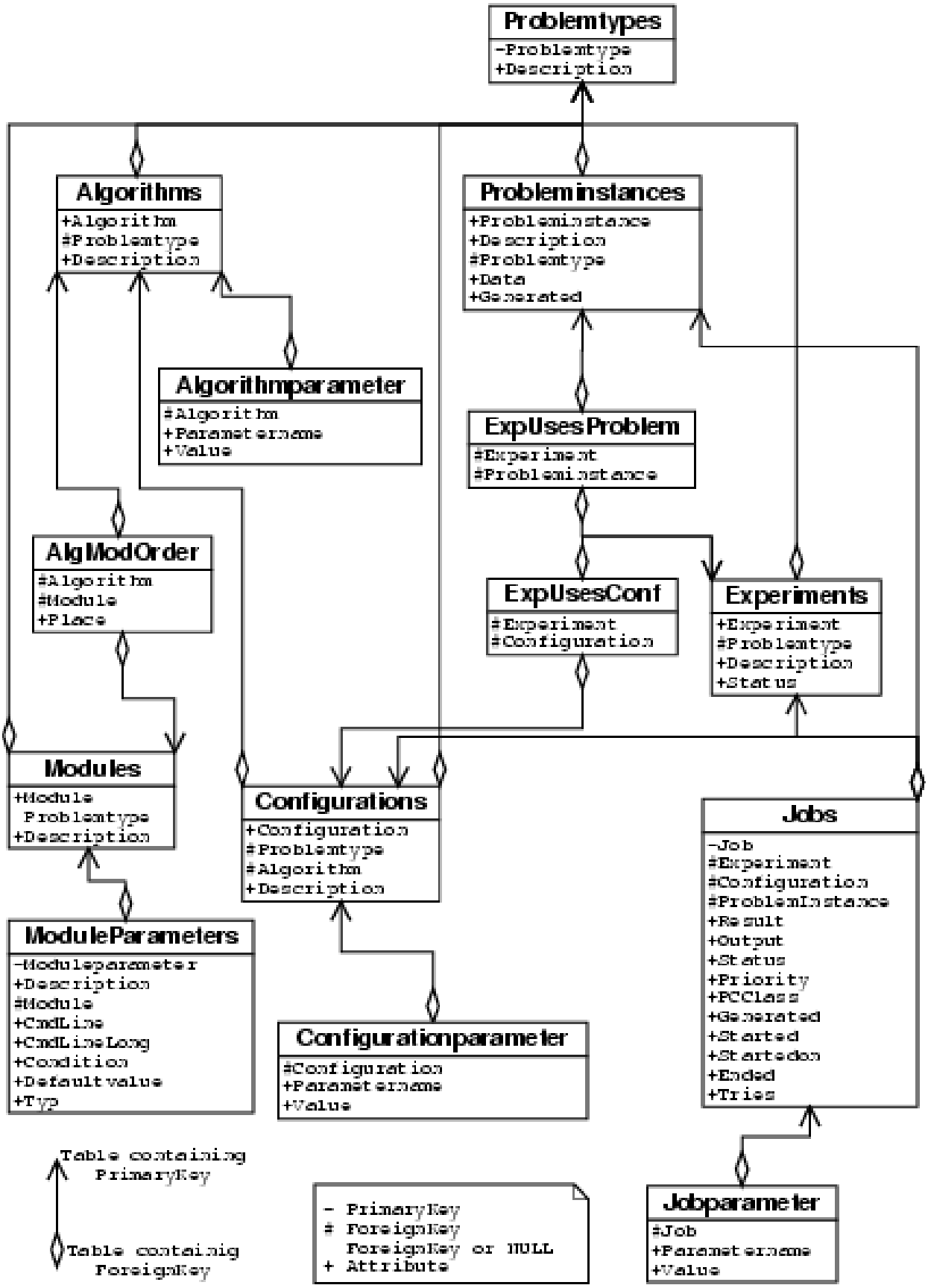


Figure 5.1: Database Structure

like. In case of a numerical value, the = operator can be replaced by any numerical comparison operator. Timestamps can be specified, too. They are specified as strings in a special format. The typical arithmetic comparison operators work for timestamps, too (see PostgreSQL documentation for more details [56]). Several attribute restriction can then be combined with SQL constructs AND and OR, which can be equipped with arbitrary brackets.

Example

For example, in order to retrieve objects of type algorithm which are used in a special experiment, i.e. when object types algorithms and experiments have to be connected, a join over tables 'Algorithms', 'Configurations', 'ExpUsesConf' and 'Experiment' must be made. Starting with table 'Algorithms', table 'Configuration' is joined with SQL command

```
Algorithms JOIN Configurations  USING ( Algorithm, Algorithm )
```

The first field or attribute 'Algorithm' is the primary key from table 'Algorithms' storing algorithms, the second one is the foreign key to table 'Algorithms' in table 'Configurations' which store any configuration. Next, table 'ExpUsesConf' is joined by appending

```
JOIN ExpUsesConf USING ( Configuration, Configuration )
```

Again, the first field 'Configuration' is the primary key in table 'Configuration', the second field 'Configuration' is the foreign key in table 'ExpUsesConf', used to reference configurations in this table. The last join is done the same way

```
JOIN Experiments USING ( Experiment, Experiment )
```

In this case, the first field 'Experiment' is the foreign key to the experiments in table 'Experiments' (storing experiments), the second field 'Experiment' is the primary key in table 'Experiments'. The complete join looks like:

```
Algorithms JOIN Configurations USING ( Algorithm, Algorithm )
           JOIN ExpUsesConf USING ( Configuration, Configuration )
           JOIN Experiments USING ( Experiment, Experiment )
```

This statement can then be used as part of an SQL SELECT statement.

If all algorithms with name A1 that have been run during an experiment named E1 *or* all algorithms with name A2 that have been run during an experiment named E2 are searched for, the following WHERE part has to be appended to the SQL SELECT statement developed just now:

```
WHERE (algorithms.algorithm = 'A1' AND experiments.experiment = 'E1')
   OR (algorithms.algorithm = 'A2' AND experiments.experiment = 'E2')
```

The SQL syntax of statements can vary. Joins can also be build using construct

INNER JOIN. Additionally, it is possibly to indicate the table relation via primary and foreign key by stating which attribute of which tables have to be identical for a tuple to be created beforehand the WHERE filter operation.

The example of just now will then look as follows

```
SELECT DISTINCT algorithms.* FROM algorithms
    INNER JOIN configurations ON algorithms.algorithm=configurations.algorithm
    INNER JOIN expusesconf ON configurations.configuration=expusesconf.configuration
    INNER JOIN experiments ON expusesconf.experiment=experiments.experiment
WHERE (algorithms.algorithm = 'A1' AND experiments.experiment = 'E1')
   OR (algorithms.algorithm = 'A2' AND experiments.experiment = 'E2')
```

Further information about the various types of joins available and about SQL queries in general can be found in [26, 35].

Typically, the search filter generation tool of the testbed does connect the object types or rather tables contributing to a query properly, so the user typically does not have to bother with building joins but can concentrate on the WHERE section to specify the attribute value restrictions and the constraints on combinations thereof: The user just has to refine a search filter. See subsection 3.5.1 on page 159 for more information about refining search filters.

5.1.2 Design Issues

The dependencies between the different object types managed by the testbed according to common database techniques suggest a database design as shown by figure 5.1 on page 229. Some additional fields, which may be redundant, have been added to improve the speed of retrieving objects by avoiding joins. The design of the database is based on the information found for example in [10], [3].

Tables 'ExpUsesConf' and 'ExpUsesProblem' depict a typical solution of how to resolve n-to-m relations between objects in a relational database.

5.2 Testbed Structure

As mentioned in chapter 2 on page 4, the testbed implementation is based on and inspired by the phpGroupWare framework. Accordingly, terms and notions used in the context of this framework are also used here. These notions together with a description of the testbed structure are described in this section, too, in addition to a detailed description of the testbed implementation and its designed structure. Many parts of phpGroupWare framework not needed yet like multi language support have been dropped for the testbed, but features can be re-added on demand with little additional effort by including the source from phpGroupWare and adopting some functions.

The first two subsections introduce several new notions that are necessary to comprehend the functioning of the phpGroupWare framework and hence the testbed. The notions are mainly related to how the source code is organized and how the user interface is decoupled from the implementation. The subsequently following three subsections are concerned with the directory structure of the source code and constitutive naming conventions which play a vital role in the functioning of the testbed. The then proximate subsections are devoted to a presentation of the most important global variables, functions and classes. This sections concludes with a short example of how the parts of the testbed work together and a subsection containing some notes that are necessary to give.

5.2.1 Applications and Services

The testbed's user interface is divided into several submenus. These submenus typically represent one major type of object the testbed is concerned with. In principle, a lot of tasks recur across these submenus, no matter which type of object they manage: Objects have to be stored, retrieved, changed, presented to the user, ex-, or imported, and so on. However, even if the tasks remain the same, they can not be performed by just one big class that handles *any* type of object. Instead, each submenu's functionality is implemented by sets of similar classes. Each such set or group of classes belonging together is called an *application*. In general, applications group together classes which somehow belong together and thus give structure to the source code which is distributed across several classes. Additionally, applications attach the parts of the implementation that are concerned with the user interface layout to the classes that implement the actual functionality. Applications typically correspond to the testbed's submenus, but exceptions are possible.

As was mentioned just now, among each application there are several recurring tasks or *services* to perform such as storing data, presenting data to the user, and so on. Consequently, several classes are used to implement the different services. A naming convention is used to distinguish different groups of *service classes* or *service objects*. The first group of service classes is concerned with the storage of data structures into the

database. It is identified by prefix `so` for `s`torage `o`bject. The classes of the second group of service classes present any data to the user and get prefix `ui` for `u`ser `i`nterface. The last group of service classes gets prefix `bo` for `b`usiness `o`bject. The classes of this group are used to provide any other functionality necessary. From now on, the different groups of service classes will also be called *so-services*, *ui-services*, and *bo-services*, respectively. Objects belonging to one of these groups are called *so-service object*, *ui-service object*, and *bo-service object*, respectively or simply service object.

The group of ui-services provides functions to check user input and to store it using an so-service if the data entered by the user was consistent. Any database interaction is handled by so-services. All functionality that has nothing to do with presenting data to or retrieving data from the user, or storing or retrieving information to or from the database typically is encapsulated in bo-services. The bo-services provide object type specific functionality like creating jobs for an experiment or running extraction scripts on the output of jobs. Additionally, bo-services are employed in a multi user environment to check whether a user has the rights to access certain data. Services of type bo can be viewed as glueing ui- and so-services together. Classifying the different services according to the Model-View-Controller design pattern (MVC),[27] the ui-services correspond to the view of the MVC design pattern, while the so- and bo-services correspond to the model part of the MVC design pattern. The so- and bo-service further subdivide the model part into a part that takes care of storage and a part that takes care of all other activities. The controller part is not very elaborated and is build in to the phpGroupWare basic framework and hence no new services had to be developed.

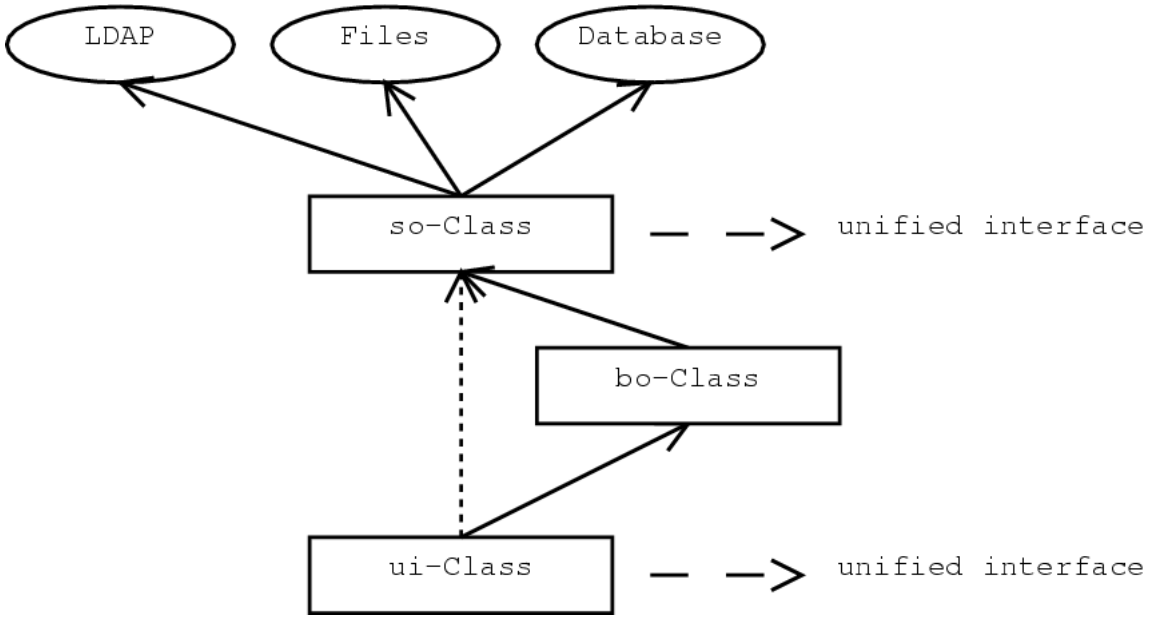


Figure 5.2: Object classes

Figure 5.2 shows which kinds of service objects use which other kinds of service objects.

Typically, ui-services use bo-services to retrieve and store data, and to perform actions like creating the jobs for an experiment. The bo-services rely on so-services for data storage and retrieval in turn. For example, a bo-service object for experiments creates and uses a bo-service object for configurations to retrieve all parameter combinations of a configuration, combines this information with the problem instance information it stores and then creates and uses a bo- and so-service objects for jobs to create and store the resulting jobs. The so-service object for jobs eventually uses the database or the file system to save the data in database tables or files. Note that currently many bo-services are omitted for many object types, since ui-services only need database access provided by so-object which they address directly.

5.2.2 Templates

The user interface of phpGroupWare is designed to be decoupled as much as possible from the actual implementation of any functionality that comes in the form of services. The motivation is to enable a change of the layout without touching the source code of the testbed. The phpGroupWare package provides a framework for developers with so called *templates* to build the user interface. A template is a file containing HTML code and *place holders*. Each page of the testbed's user interface is encoded by one or more templates. The template's HTML code represents the general layout of the page while the place holders represent the variable and changing information. The place holders are filled out by ui-service objects which might use other service objects during this task, e.g. for retrieval of information. Place holders are indicated by brackets '`{`' and '`}`' in the template files.

Typically, when the user triggers some action such as creation, deletion, or simply change of submenu, a function of the responsible ui-service is called. This function knows which template it must use and hence which page layout is presented to the user. For example, if the user requested to view the 'Experiments' submenu overview, function `GetList()` of the class implementing the ui-service for experiments (which is `experiments.uiexperiments`) gets called. This functions knows which layout is required to present a list of experiments and how to fill the needed information. The layout is chosen using function `set_file` which sets any necessary template files. This function can take an array of files as argument. Next, all necessary steps needed to compute or retrieve the information to be filled into the template as place holder replacement, possibly with the help of other services, are performed. Finally, the replacement is filled in by assigning it to place holders with function `set_var` which takes as first argument the name of the place holder and as second the value, which can be any valid HTML code. The phpGroupWare framework takes care of all the rest: Loading the template, displaying it to the user, actually filling the place holders and calling the right function when yet another action was triggered by the user, repeating the process described just now.

5.2.3 Directory Structure of the Testbed

The directory structure of the testbed containing the source code reflects the grouping of the testbed classes in the form of applications: Each application has exactly one subdirectory assigned. Referring to the notions used in chapter 3 on page 50, the location of the testbed's main source code directory is `TESTBED_ROOT`/*testbed*. The available applications together with the functionality they provide are described next:

probleminstances

This application corresponds to the 'Problem Instances' submenu. It features ui- and so-services and contains the necessary templates to display problem instances. Problem instances can be ex- or imported, viewed, copied edited, deleted and newly created. Each such functionality has a corresponding function in the ui-service class that implements it. The elementary database access tasks such as insert, delete, etc. are implemented by individual functions in the so-service class. This holds for all other applications that coincide with main object types of the testbed, too. As is the case for all applications, too, the ui-service is responsible for any gimmick of the user interface that enhances usability such as expansion and implosion of cells or complete columns of tables.

modules

The representation of modules is managed by this application. Note that the modules themselves do not exist in the testbed (database) but reside in the executables subdirectory of the testbed. However, a description of any module registered is available. Modules or rather their representations can be viewed and the description can be edited. This application only features ui- and so-service classes and some templates.

algorithms

Algorithms can be ex- or imported, viewed in detail, copied, deleted, its description can be edited and they can be created. This application contains ui- and so-service classes to do so together with the necessary templates. The so-service class of this application takes care that an algorithm specification – including hidden parameter, order, kind, and number of modules, and so on – is stored in the database properly.

configurations

Application 'configurations' handles configurations. All operations that can be performed on algorithms as described just now can be performed on configurations, too. Therefore, it provides the same ui- and so-services and the appropriate templates. The services of this application additionally take care of range checking

(compare to paragraph 'Parameter Specification of Modules' on page 15 in subsection 2.3.1 and to subsection 4.2.1 on page 181) and computation of the fixed parameter settings in dependence on the parameter values and conditions entered.

experiments

This application is responsible for all actions related to experiments. It features ui- and so-services and a number of different templates. The actions available for experiments are the same as those for algorithms and configurations. The so-service class is responsible for computing the current status of an experiment and of generating the jobs of an experiment. This is done using services from the 'jobs' application. The ui-service is responsible for assigning priorities and hardware classes to jobs (compare to subsection 3.3.8 on page 116 and subsection 3.3.14 on page 134).

jobs

Jobs can be started and restarted, their output to standard output and to the result file can be viewed, they can be canceled, suspended, and resumed as is listed in table 3.6 on page 124. This application contains all groups of services. The ui-service class conditions all information presented to the user. Any retrieval or storage of data including any necessary formating or further computation on data is accomplished by the so-service. The bo-service class is concerned with any tasks related to running a job such as retrieving the next jobs from the job execution queue, retrieving the corresponding parameters and their settings, actually running the binary, updating the output storage in the database, and so on. Any job server (compare to subsection 3.4.4, 3.3.8, or 3.3.9) will only use functions of the bo-service class of this application to perform its task. It does not implement any substantial functionality with respect to running a job itself.

statistics

This application groups together anything related to the statistical analysis of an experiment. These are foremost data extraction or analysis scripts and their application to job results. For each data extraction and analysis script objects, separate service classes exist. Services of type ui are responsible for presenting all scripts to the user, as well as editing, deletion, viewing, im- or export, copying, and creation of scripts. Additionally, they provide pages for specifying and running a data extraction or analysis effort as was described in subsections 3.3.10 and 3.3.11 on pages 125 and 129, respectively. Finally, ui-services of this application are concerned with managing the download of any data extracted. Any storage and the actual im- or export of scripts is done by so-services, while bo-services are used to actually execute the scripts to extract data or to perform the statistical analysis. These bo-services translate any macros in scripts, retrieve any necessary

jobs results from the database with the help of the so-service for jobs and address the R engine in order to perform the actual statistical tests or plots.

common

Any object type or other functionality not listed yet is handled by the classes centralized in application 'common'. This application comprises service and templates for problem types, hardware classes, categories, basic database access, debugging, global functions, search filter actions and some minor further functionality. The ui-service and templates for search filters provide the search filter generation mask and, together with the basic database service, generate a final SQL statement implementing a search filter. Services of type ui and appropriate templates provide all necessary functionality for categories, hardware classes and problem types. This application also contains any java scripts used throughout the testbed and the parent class of all module definition files implementing basic behavior common or rather mandatory for any module definition file. Finally, this application concentrates any commonly used images.

Additional to the application directories, other subdirectories contain source code implementing the command line interface tools of the testbed and the database specification. These are described in the following:

bin

This subdirectory contains in file *cmd.php* the code for the command line tool of the testbed implementing any functions as discussed in section 3.4 on page 136.

database

Any database scripts are contained in this subdirectory. These scripts are used to set up an initial database structure when installing a new testbed (compare to section 3.1 on page 51) or to upgrade a database when the testbed is updated and potentially the database structure has to be changed slightly. Database scripts are indicated by subfix `.sql`.

devel

This subdirectory contains the tools for automatic generation of module definition files (compare to subsection 4.2.1 on page 181). Additionally, a template module definition file which is filled during the generation process is located here.

manual

The testbed provides online help in the form of this manual as HTML or PDF file and two special help pages for help with regular expressions and conditions when configuring an algorithm. The according files containing the help information and the manuals are located in this subdirectory.

5.2.4 Naming Conventions for Class Names

All objects are created with the same global function named `CreateObject`. This function takes as argument the name of a class. The resulting object can then be used to execute its member functions. This can be done directly with function `ExecMethod`, too, which takes the name of a class and a function name, separated by a dot '.', as argument.

PHP is an interpreted programming language. It does not precompile any code and does not now where to find any code in advance. The problem faced by the two function mentioned just now is how to find the appropriate code, i.e. the appropriate class definition given just its name. This problem is solved by the phpGroupWare framework by using a strict naming schema for any class and function which automatically provides information about the location in the form of which application a class belongs to and which type of service it implements.

Each class name has the same structure: `<service><type>`. The type of service class is indicated by part `<service>` which can be `so`, `ui`, or `bo` for so-services, ui-services, and bo-services, respectively. Part `<type>` of the class name represents the object type the class is concerned with. If no special type of service or object type is applicable such as for the class containing globally used functions (which is `functions`), the service part `<service>` is omitted and the type part `<type>` is used to name the class according to its purpose. Each class resides in a file named `class.<classname>.inc.php` where `<classname>` is equal to the before mentioned `<service><type>`. This naming convention, however, makes class names not unique across applications. Therefore, whenever a class is referred to, e.g. when creating it, the class name itself is prepended by the name of the application it belongs to, separated by a dot '.'. With `<appname>` indicating the name of the application this yields class names used for creation as follows: `<appname>.<classname>`. A statement `CreateObject('<appname>.<classname>')` then automatically leads the interpreter to the appropriate source code with is located in the subdirectory of application `<appname>` and among the files implementing service classes there, file `class.<classname>.inc.php` contains the source code.

Example:

When the user wants to see all algorithms contained in the testbed, it switches to the 'Algorithms' submenu by clicking on the corresponding link in the testbed's main menu. This link will trigger function `GetList()` of class `uialgorithms` as indicated by the link name: *http://localhost/testbed/index.php?_menuaction=algorithms.uialgorithms.GetList*. Such a link contains information about the application, the class name and the function to use and hence the code can be retrieved and executed. In the course of the execution of function `GetList`, the ui-service object for algorithms created before needs to retrieve the algorithms from the database to display them. In order to this, it creates the correspond-

ing so-service object with command `CreateObject('algorithms.soalgorithms')`. Again, the application and the class name suffice to find the source code. The ui-service can now use any function of the newly created so-service object, such as retrieving all algorithms contained in the testbed, to fulfill its task.

5.2.5 Directory Structure of an Application

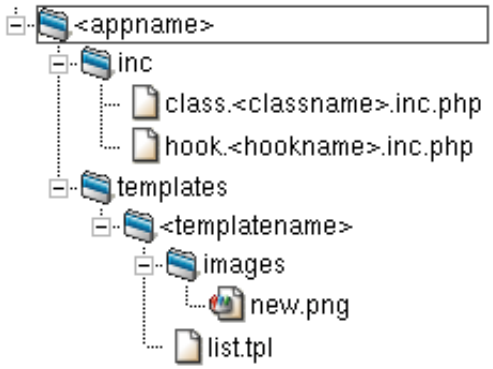


Figure 5.3: Directory structure of an application

Each application not only contains service classes but also templates for user interface layout and images to be displayed on the web pages such as the symbols for icons triggering actions on submenu entries. Thus, the directory structure of an application is further extended in a systematic way based on the directory structure phpGroupWare is using and which hence is part of the design, since the PHP framework interpreter searches this directory structure to find the appropriate service classes. Changing the hierarchy can only be done on the expense of changing all affected class names, too. This subsection discusses the directory structure of an application: Important mandatory directories, their required standard names and their function in the testbed functioning are listed and described here. In the following, `<appname>` denotes an arbitrary but fixed string used as name of an application, while `<classname>` denotes an arbitrary but fixed string used as name of a service class (including the service class group prefix). The notions used in chapter 3 on page 50 to locate the testbed's main source code directory `TESTBED_ROOT`/*testbed* are adopted.

- `TESTBED_ROOT`/*testbed*/`<appname>/inc/class.<classname>.inc.php`

  All service classes of an application are located in this subdirectory of an application directory. Functions `CreateObject` and `ExecMethod` will look here to find the source code.

- `TESTBED_ROOT`/*testbed*/`<appname>/inc/hook.<hookname>.inc.php`

Hooks enable an application to react on actions or rather triggers from other applications, if the other application is calling such a hook. For example, an application can register the import handler for XML data by providing a hook named `XMLExchange`. Hooks are functions that are called when the action implemented by the hook is supposed to be triggered. Inside the `XMLExchange` hook, for example, the application registers XML tags for parts of the data that was exported previously by the application, thereby using an object with function `ImportXML`. Hooks are invoked by other applications by calling `$GLOBALS['testbed']->hooks ->Run('<hookname>');`.

- `TESTBED_ROOT`/*testbed*/`<appname>/templates/<templatename>`

  As mentioned before, web pages are build out of templates. Each application provides its own templates which are stored in files. For different actions like viewing data, creating new data, and so on, different templates are provided. Templates for one single web page can be spread over different files, too. Services of type ui extract the data needed to fill a template, so the developer need only decide where to store the information extracted in the template HTML code. The phpGroupWare framework additionally supports different so called *template schemes* which the user can select. Each user can in principle design its own look of an application by writing its own scheme in the form of a collection of template files stored in a subdirectory of the template directory of the corresponding application. Such a subdirectory is indicated here by `<templatename>`. At the moment, for each application only one template scheme is implemented yet which is called `default`. The `default` template directory is searched for a template file, if the search for an user defined template file failed. With the help of templates the appearance of the search mask in submenu 'Search filter' can be changed, for example, as is explained in subsection 5.3.2 on page 270.

- `TESTBED_ROOT`/*testbed*/`<appname>/templates/<templatename>/images`

  Images (e.g. for icons) might change with different template schemes. Hence, they should be stored in a special image subdirectory of a template scheme directory. Class function `$GLOBALS['ui']->image('<appname>','<imagename>')` (compare to subsections 5.2.6 and 5.2.9 on pages 241 and 250, respectively) stores a function that can return a complete and proper HTML link to image `<imagename>` of application `<appname>` for use in a template. Image name `<imagename>` is used without the suffix of the graphics format. This suffix will be added automatically by function `image`. Images used by the current application are first searched for in the image subdirectory of the active template scheme directory of the current application, then in the image subdirectory of the default template scheme `default` of the current application, and last in the images subdirectory of application `common`. The advantage of this procedure is that a developer can use individual icons instead of the standard icons from the testbed by just placing corresponding images

in the appropriate application image subdirectory.

5.2.6 Important Environment Variables

This subsection discusses the most important globally visible environment variables – embedded in nested data structures – the testbed creates or evaluates. In order to view the value of a variable during execution of testbed, variables can be printed to the top of most testbed web pages by either using command `echo` followed by the variable name and a semicolon, for example `echo $variable;`, or by using command `print_r` with the variable name as argument in case the variable is an array, for example `print_r($_SESSION);`. The structure of a variable can also be viewed within the testbed by appending a line `dump2(<variablename>);` to file `TESTBED_ROOT/index.php` (e.g. `dump2($_SESSION)`).

`$GLOBALS['flags']`

Flags are used by the testbed to determine in which environment the testbed is running. Depending on the environment, it then can provide appropriate services and functions. The flags employed by the testbed are described in detail next. The type of each flag is written in italics and round brackets behind its name.

- `$GLOBALS['flags']['ui']` *(string)*

  This flag determines which type of user interface is used. Depending on this flag different classes are used within the testbed to provide basic user interface (ui) functions. This flag must be set by an application of the testbed framework. The value can either be

  - `web`: The testbed is used with a browser via an HTTP-Server,
  - `console`: The testbed is used on the console or rather command line, or
  - `none`: No user interaction is needed at all.

- `$GLOBALS['flags']['currentapp']` *(string)*

  This flag defines the currently active application.

- `$GLOBALS['flags']['template']` *(string)*

  This flag holds the name of the currently active template scheme. It is set automatically by the testbed by retrieving the settings from the user environment variable which will be presented next. At the moment only one template (`default`) is available and set automatically by the testbed.

`$GLOBALS['user']`

This variable stores information related to a testbed user to enable access to this information from several HTML pages. Typically, this is the session data of the user using a web front end. It contains the same data as variable `$_SESSION['user']` or `$GLOBALS['HTTP_SESSION_VARS']['user']`. At the moment, this structure does not contain many information. It is mainly concerned with some settings the user can make in the 'Preferences' submenu (see subsection 3.3.12 on page 132). The default values for settings described here are set upon testbed start in configuration file `TESTBED_ROOT`/*config.php*. They can be set in the user specific configuration file, too (see subsection 3.1.4 on page 69 for more information about configuring the testbed).

- `$GLOBALS['user']['preferences']['common']['maxmatchs']` *(integer)*

  This part of the user information stores the maximum number of entries to show on one page.

- `$GLOBALS['user']['preferences']['common']['textrows']` *(integer)*

  The number of rows of text input fields is stored here.

- `$GLOBALS['user']['preferences']['common']['textcols']` *(integer)*

  The number of columns of text input fields is stored here.

- `$GLOBALS['user']['XMLExport']` *(array used by class common.XMLExchange)*

  This array determines which object types are automatically exported, too, when an object is exported via XML.

- `$_SERVER['REMOTE_USER']` contains the login of the remote user that is using the web front end right now. This variable's contents can be used to get information about the current user such as the location of its home directory. The home directory then can be scanned for the user specific configuration file (*~/.testbed.conf.php*) in order to access the correct database. In a multi-user setup of the testbed it is mandatory that the user information can be accessed. To do so, the web server must identify any user first using an authentication mechanism such as LDAP[1]. Hence, such a mechanism must be established and the web server has to be instructed to use it by creating an appropriate file `.htaccess` in the testbed root directory `TESTBED_ROOT`. For more details see section 3.1 on page 51 and specifically subsection 3.1.4 on page 69.

[1]Lightweight Directory Access Protocol

`$GLOBALS['testbed']` *(object)*

Services and objects used frequently are automatically created by the testbed so the developer does not need to create them directly. These objects can be accessed by:

- `$GLOBALS['testbed']->db` *(object)*

  Access to the global database object of the testbed used for any actual database interaction.

- `$GLOBALS['testbed']->ui` *(object)*

  Access to the object that handles user interface functions.

- `$GLOBALS['testbed']->hooks` *(object)*

  Access to the service that executes hooks.

- `$GLOBALS['testbed']->template` *(object)*

  Access to the template object for usage within the 'common' application (e.g. class common.nextmatchs which is used to limit the number of entries in a submenu depends on it).

- `$GLOBALS['testbed']->cats` *(object)*

  Access to the object which provide access to the categories for the current application.

`$GLOBALS['ui']` *(object)*

This variable is a shortcut for `$GLOBALS['testbed']->ui`

5.2.7 Session Variables

The HTTP[2] protocol itself has no state. That is, any information which was entered by a user on a web page can not be stored by the page itself. Instead, so called *sessions* are used to keep states and information over different pages a user visits. The state or session information is handled by the browser used to display the HTTP pages and will be lost after the user has closed its browser. Additionally, for each browser employed by a user, different and independent session information will be kept. Each user of the testbed has its own session state. There are possibilities to keep information such as login information and user preferences between session. However, at the moment, this functionality is not implemented.

[2]Hyper Text Transfer Protocol

Session variables store state information and are described in what follows. Note that all session variable names begin with prefix `$GLOBALS['HTTP_SESSION_VARS']` (in PHP above version 4.1 also with `$_SESSION`). This prefix of the variable names will be omitted in the following descriptions. Again, the type of a variable is given in round brackets and italics.

- `['problemtype']` *(string)*

  The selected default problem type is stored in this session variable.

- `['message']` *(string)*

  The text in this variable will be shown on the next upcoming page, which uses the `['ui']->Navbar()` function (see description of class ui). Messages are useful to give the user some feedback that an action like delete, insert or edit was successful or failed.

- `['history']` *(array)*

  This structure is used to keep track of the pages that were visited by the user. That way, it is possible to lead the user automatically back to the previous pages after some operation was done. For example, the user is automatically lead back to the last page visited before after having canceled a deletion of a problem instance. This structure is mainly used by class `common.ui` which takes care of most basic user interface tasks.

- `['filters']` *(array of strings indexed by application names)*

  This structure contains the active current search filter for each application. For example, the current search filter for jobs is accessed by `['filters']['jobs']`.

- `['appsession']` *(array of mixed types[3] indexed by application names)*

  In this structure, each application can store the information that it needs to be kept between pages. The structure can be defined by each application individually. For example, application 'configurations' use `['appsession'] ['configurations'] ['nm']` to keep the information of what is displayed on the overview page of the 'Configurations' submenu.

- `['user']`

  This structure has already been described in subsection 5.2.6 on page 242.

[3] A mixed array can have arbitrary types as values, even other arrays. Which is used depends on the application.

5.2.8 Global Functions

A lot of functions are class independent and frequently and globally used. These are centralized in file `functions.inc.php` in directory `TESTBED_ROOT`*/Testbed/common/inc* and described here. The documentation of global functions is quite brief. For detailed information about the mode of operation of a function, the implementation and its additional comments should be consulted. Before starting to explain the most important global functions, some notions used in the class and function descriptions that come next are explained.

`FORM`

FORM indicates that the context used is a HTML form element. All input that is entered on a web page is put in a FORM (formula) and send to the web server for evaluation. Input fields, check boxes, selection lists and the like are part of a FORM.

`GET`

A FORM element can transfer the information entered to the server with two methods, GET and POST. Method GET appends the data entered to the call address which then can extracted by the server.

`POST`

If using method POST (= send) to transfer data entered into a form, this data is provided through the output input channel by the server. A special script is necessary that treats the data as ordinary user input as if it were run on the command line. This method is required, if the data entered is extensive.

`&$<parametername>`

This type of function argument specification indicates that a parameter is called by reference instead of a call by value (which is the PHP default behavior).

`$this`

This PHP language construct indicates an object itself.

Filenames include a path that is relative to/starts at directory `TESTBED_ROOT`*/Testbed/*.

Collection of Direct/Global Functions

Abstract: Provides all functions which are required to be availble at the lowest level.
Description: Direct or global functions implement basic functionality.
File: common/inc/functions.inc.php

- **function sanitize**
  `sanitize($string,$type)`
  Abstract: Validate data of different types.
  Description: This function is used to validate input data for a given format.
  Parameter (1) `$string`: String containing input data to check.
  Parameter (2) `$type`: Data type or format that is checked for.
  Result: Boolean. True, if and only if the first parameter matched the given format.
  Example:
  `sanitize($somestring, 'number');`

- **function CreateObject**
  `CreateObject($class, $p1='_UNDEF_', .. ,$p16='_UNDEF_')`
  Abstract: Create an object given a class name and include the class file if not already done.
  Description: This function is used to create an instance of a class given by its name in the form of a string. If the class file has not been included yet, it will be. The name of the class must be prefixed with the application name where the class can be found separated by a dot: '.'.
  Parameter (1) `$class`: String containing name of class (including the application name).
  Parameter (2) `$p1`: -`$p16`: Class parameters (all optional).
  Result: Newly created object.
  Example:
  `$html = CreateObject('common.html');`

- **function ExtractFormVars**
  `ExtractFormVars()`
  Abstract: Extract values of variables that have been submitted by a FORM.
  Description: This function extracts any values of variables from a submitted FORM. All variables in POST and GET that start with a '_' are omitted. All other variables that contain a '_' are split by the '_' and transformed into an array. A name like 'a_b_c' will be transformed in $result['a']['b']['c'], $result being the resulting array.
  Result: Array with the data of the form.

- **function ExecMethod**
  `ExecMethod($method, $functionparams = '_UNDEF_', $loglevel = 3, $classparams = '_UNDEF_')`
  Abstract: Execute a function of an object.
  Description: This function is used to create an instance of a class and execute the function of that object. It is also possible to execute objects that are part of other objects.
  Parameter (1) `$method`: String with identifier of function to execute. The function

identifier must contain the application, class name, and function name separated by dots '.'. Compare to function CreateObject.
Parameter (2) `$functionparams`: Parameters for the function in the form of an array.
Parameter (3) `$loglevel`: Developers choice of logging level.
Parameter (4) `$classparams`: Parameters to be sent to the constructor of the class.
Result: Result of the executed method. This can be any type.

- **function CreateModule**
  `CreateModule($module)`
  Abstract: Create an instance of an object which represents a module definition file.
  Description: This function is used to create an instance of a module definition file object. The module definition file is included if not already done.
  Parameter (1) `$module`: Name of module definition file.
  Result: Object representation of the module definition file.
  Example:
  ```
  $obj = CreateModule('filecopy');
  ```

- **function dump**
  `dump($arg)`
  Abstract: Dump a PHP array into a readable form to a the string that is returned.
  Parameter (1) `$arg`: Array to dump.
  Result: String containing a readable version of the argument array.

- **function AddVarname2array**
  `AddVarname2array(&$dst, $name, $value)`
  Abstract: Add a variable name with '_' to the destination array.
  Description: First, name $name is split by '_' as described for function ExtractFormVars. The individual components are then added the multidimensional array $dst. If array $dst does not yet exist, it is created on the fly. Any existing parts will be overwritten. This function is used in the context of adding or extracting data to or from FORMs (compare to function ExtractFormVars).
  Parameter (1) `&$dst`: Array to add the variable to.
  Parameter (2) `$name`: String containing the name of the variable to add.
  Parameter (3) `$value`: Value of the variable to add.
  Example:
  ```
  test=array()
  AddVarname2array($test, 'foo_bar_test', 42);
  $test == array('foo' => array('bar' => array('test' => 42)))
  ```

- **function MakeFlatArray**
  `MakeFlatArray($data)`
  Abstract: Transform a multidimensional array into a flat array.
  Description: If the array is multidimensional, the array slice are transformed to fit into one flat array. This is done by joining recursively the key names with a '_'. It is often used to preserve data structure in FORMs. Function ExtractFormVars automatically transforms this structure back to an array. This function is used in the context of adding or extracting data to or from forms (compare to function ExtractFormVars).
  Parameter (1) `$data`: Array to transform.
  Result: Flat array with the transformed data.
  Example:
  `MakeFlatArray( 'foo' => array('k1'=>'v1'), 'bar' => 'test') ) == array('foo_k1'=>'v1', 'bar' => 'test').`

- **function Status2Text**
  `Status2Text($no)`
  Abstract: The status number $no used by the testbed to indicated the statuses of jobs and experiments is returned as readable string.
  Parameter (1) `$no`: Status number to transform.
  Result: String with a readable version of the status.

- **function lang**
  `lang($text)`
  Abstract: Translate string $text to the user defined language.
  Description: This function is used in phpGroupware, and may also be used to make the testbed multilingual. For now, any translation mechanism has been stripped from the phpGroupWare framework and hence the testbed, because it is not needed. It can be reimplemented by changing this function and including some classes from phpGroupWare.
  Parameter (1) `$text`: Text to translate.
  Parameter (2) Optional additional arguments are to be placed in the string.
  Result: Translated string and possibly some options set.
  Example:
  `lang('Showing %1 out of %2', 21, 42) == 'Showing 21 out of 42'`

- **function strip_html**
  `strip_html($s)`
  Abstract: Convert HTML special characters contained in a string to appropriate HTML tags.
  Parameter (1) `$s`: String to transform.
  Result: Transformed string which can be used directly in HTML code without breaking the layout.

- **function get_account_id**
  `get_account_id($id = 0)`
  Abstract: Returns the account ID of the current user.
  Parameter (1) `$id`: Account name of a user.
  Result: Integer with the ID of the current user.

- **function get_var**
  `get_var($variable,$method=array('GET','POST'),$default_value='')`
  Abstract: Retrieve a value from either a POST, GET, COOKIE, or other FORM variable.
  Description: This function is used to retrieve a value from a user defined variable within an ordered list of methods.
  Parameter (1) `$variable`: Name of the variable.
  Parameter (2) `$method`: Ordered array of methods to search for supplied variable.
  Parameter (3) `$default`: _value (optional) If no value was found, this default value is returned.
  Result: Value of the searched for variable.

- **function array2xml**
  `array2xml($data, $cdata = array() )`
  Abstract: Transform an array into an XML compliant string.
  Description: This function transforms a string into XML compliant code.
  Parameter (1) `$data`: Data to transform into XML code.
  Parameter (2) `$cdata`: Array with field names that must be escaped in XML CDATA tags.
  Result: String with the XML tags.

- **function array_cmp**
  `array_cmp($a ,$b)`
  Abstract: Compare two multidimensional arrays for whether they contain the same values and keys, i.e. whether they are equal.
  Parameter (1) `$a`: First array to compare.
  Parameter (2) `$b`: Second array to compare.
  Result: Boolean: True, if and only if the two arrays are equal in the just mentioned meaning.

- **function myarray_diff**
  `myarray_diff(&$a ,&$b)`
  Abstract: Remove same elements from two multidimensional arrays, leaving only different elements.
  Description: The function does not return a result, because all changes are made on the arrays given as parameters directly which are provided as references. The arrays given as parameters may only be used afterwards to show the differences,

but may not be used further to store the left information to a database.
Parameter (1) `&$a`: First array.
Parameter (2) `&$b`: Second array.

- **function cleanupTemp**
  `cleanupTemp()`
  Abstract: Remove temporary files from the testbed that have not been deleted for unknown reasons.

- **function prettySQL**
  `prettySQL($string)`
  Abstract: Formats an SQL statement into a pretty formatted string.
  Description: This function uses the sqlparser from phpMyAdmin to bring an SQL statement into a mor human readable form with tabbing and highlighting. Used for displaying categories.
  Parameter (1) `$string`: String containing the SQL statement.
  Result: String with the CSS (Cascade Style Sheets) formatted pretty SQL statement.

5.2.9 Basic Classes

After the description of global variables and functions, a brief description of the most important classes follows. Service classes are not described here, since their functioning was sketched before (see subsection 5.2.1 on page 232). Any details of how individual service classes work must be looked after in the corresponding source code and its comments.

The classes described here implement basic operations to present data to the user, to access the database or to perform other common tasks. The documentation the classes is quite brief. For detailed information about the mode of operation of a function, the implementation and its additional comments should be consulted.

Class testbed

Abstract: This class is used to initialize some services and functionality of the testbed.
File: common/inc/class.testbed.inc.php

- **function get_db_obj**
  `get_db_obj()`
  Abstract: An object to access the database of the testbed is returned.
  Description: This function must be used to retrieve a database object which handles any access to the database. This is because only this way the global database

transaction mechanism can be guaranteed to work properly. If a database object is needed that connects to another database, this new database object can be created with CreateObject('common.db'). It does not need to be created with this function.
Result: Database object connected to the database as set by the user's configuration file.

- **function registerVersion**
  `registerVersion($app, $versionString)`
  Abstract: Register a version of an object or service used.
  Discussion: This function is used to keep track of used services and objects mainly for debugging purposes.
  Parameter (1) `$app`: Application the object stems from.
  Parameter (2) `$versionString`: CVS version string.

- **function getVersion**
  `getVersion()`
  Abstract: Return the current version of the testbed.
  Result: String with the current version information.

Class ui

Abstract: This class provides a lot of basic functions that are repeatedly used by many functions and objects to display parts of the web front end user interface, i.e. the web pages.
File: common/inc/class.ui.inc.php

- **function Navbar**
  `Navbar()`
  Abstract: Print the HTML navigation bar, each application can then later print its specific parts.
  Description: Only applications that use this function to print on the screen will be kept in the navigation history. It also prevents a repeated print of the navigation bar, e.g. if one application calls another application to display some parts. .

- **function Footer**
  `Footer()`
  Abstract: Print the HTML Footer after an application has printed its parts of the user interface.

- **function Headline**
  `Headline( $text )`

Abstract: Print a headline to the screen.
Parameter (1) `$text`: String containing the text to print as headline.

- **function Subline**
  `Subline( $text )`
  Abstract: Print a subline to the screen.
  Parameter (1) `$text`: String containing the text to print as subline.

- **function SetPage**
  `SetPage( $url )`
  Abstract: Add a URL to the history.
  Parameter (1) `$url`: URL to visit, if function LastPage() is called.

- **function LastPage**
  `LastPage()`
  Abstract: Direct the browser to the last page in the history, i.e. the last page visited.
  Description: The page is visited and thereby removed from the history.

- **function RemoveLastPage**
  `RemoveLastPage()`
  Abstract: Remove the last page from the history.

- **function Location**
  `Location($url)`
  Abstract: Direct the browser to an URL.
  Description: The history is not influenced by this function.
  Parameter (1) `$url`: URL to direct the browser to.

- **function Halt**
  `Halt()`
  Abstract: Print a footer and exit the execution of the script.

- **function strip_html**
  `strip_html($s)`
  Abstract: Remove all special characters that are used in HTML like <,>, and so on from the string given as argument.
  Parameter (1) `$s`: String to remove special characters from.
  Result: String without special characters.

- **function link**
  `link($url, $params = false)`
  Abstract: Generate a link within the testbed (path).
  Description: The link is prepend with the URL where the testbed is located. The developper does not need to care about the absolute link (path) and can assume

that the testbed is run in its own web server root directory and all links are made relative to that root. The appropriate prefix of the URL is added by this function. Some additional parameters are added automatically, if they are need such as (e.g. for the session management).
Parameter (1) `$url`: URL link to be completed.
Parameter (2) `$params`: Parameters to be added to the URL
Example:
`$this->link('/index.php','_menuaction=app.class.func') ==`
`'http://host/testbed/index.php?_menuaction=app.class.func'`

- **function helplink**
  `helplink($topic, $app='common', $mark='')`
  Abstract: Generate a link that opens a help window.
  Parameter (1) `$topic`: String referring to the HTML file containing the topic text to show.
  Parameter (2) `$app`: String containing the name of the application that employs the help link.
  Parameter (3) `$mark`: String holding on to a special mark within the topic text that is to be shown when the window opens.
  Result: String containing the created link.

- **function helpbutton**
  `helpbutton($topic, $app='common', $mark='')`
  Abstract: Generate a button that opens a help window.
  Parameter (1) `$topic`: String referring to the HTML file containing the topic text to show.
  Parameter (2) `$app`: String containing the name of the application that employs the help button.
  Parameter (3) `$mark`: String holding on to a special mark within the topic text that is to be shown when the window opens.
  Result: String containing HTML code for the button.

- **function image**
  `image($app, $name)`
  Abstract: Generate a link to an image within an application.
  Description: Depending on the used template scheme, the image is searched for in different image directories.
  Parameter (1) `$app`: String containing the name of the application that contains the image.
  Parameter (2) `$name`: String containing the name of the image.
  Result: String with the URL of the image.

- **function menuimage**
  `menuimage($app, $name)`
  Abstract: Generate an HTML image tag with a description.
  Parameter (1) `$app`: String containing the name of the application that contains the image.
  Parameter (2) `$name`: String containing the name of the image (also used for a description of the image).
  Result: String with the HTML image tag.

- **function imagelink**
  `imagelink($app, $name, $url, $params)`
  Abstract: Generate a link image whose link points to the given URL.
  Description: Shortcut for the combined usage of menuimage() and link()
  Parameter (1) `$app`: String containing the name of the application where the image can be found.
  Parameter (2) `$name`: String containing the name of the image.
  Parameter (3) `$url`: URL within the testbed.
  Parameter (4) `$params`: Parameters of the URL.
  Result: String with the HTML image link to the specified URL.

Class db

Abstract: This is the database class whose instances solely communicate with the testbed's ProstgreSQL database.
Description: This class can be replaced by a class that can operate with other databases like MySQL by providing the same interface as this class.
File: common/inc/class.db.inc.php

- **function db**
  `db($settings = '')`
  Abstract: Constructor of the class.
  Parameter (1) `$settings`: Settings how to connect to the database. These settings typically stem from the user or the global configuration file.

- **function connect**
  `connect()`
  Abstract: Connect to the database.

- **function to_timestamp**
  `to_timestamp($epoch)`
  Abstract: Convert a date to the format the database requires for dates.
  Parameter (1) `$epoch`: Contains the date to convert.
  Result: String with the correspondig database date format

- **function from_timestamp**
  `from_timestamp($timestamp)`
  Abstract: Convert a database date to a PHP date.
  Parameter (1) `$timestamp`: Timestamp from the database.
  Result: A PHP date.

- **function disconnect**
  `disconnect()`
  Abstract: Disconnect from the database.
  Discussion: This only affects systems not using persistent connections.
  Result: Boolean. True, if and only if the disconnect was successful.

- **function db_addslashes**
  `db_addslashes($str)`
  Abstract: Add slashes to escape characters "' or "" in a string stored in the database.
  Parameter (1) `$str`: String to escape.
  Result: String with the escaped special characters.

- **function query**
  `query($Query_String, $line = '', $file = '')`
  Abstract: Run a query on the database.
  Description: This function runs a query on the connected database. If the connection has not established yet, this will be done automatically now. If the global debug option is set, a history of the queries is stored in variable $this->QueryHistory. Also, a unified query is stored to a file (all strings are replaced with 'x' and each number is stored as '0'). Such it is possible to get an overview over which sorts of queries are run and how often. This can help to optimze the database.
  Parameter (1) `$Query`: _String String containing the query to run in the form of an SQL statement.
  Parameter (2) `$line (default '')`: Line of code which requested the query (for debugging purposes).
  Parameter (3) `$file (default '')`: Source code file where the query was requested (for debugging purposes).
  Result: Boolean. True, if and only if the query was succeful.

- **function make_query**
  `make_query($table, $base, $struct, $defaultorder )`
  Abstract: Generate queries based on search filters or nextmatches constraints (which restrict the number of entries that are displayed in a submenu) in order to select the exact number of objects and the overall number o objects in case of nextmatches constraints.
  Parameter (1) `$table`: String denoting the base table to run the query on.

Parameter (2) `$base`: String containing the SQL query without any limits and filters.
Parameter (3) `$struct`: Structure containing all information about the filters or nextmatches constraints.
Parameter (4) `$defaultorder`: Default order for how the selected entries should be ordered.
Result: Array with two fields containing the query (i.e. SQL statement) that will retrieve the entries to actually show and a query that will get all possible objects and which can be used to computes the overall number.
Example:

```
makey_query("test", "SELECT * FROM test", array(
'test.fieldname' => 'condition'), "fieldname" );
```

- **function db_string**
  `db_string($value)`
  Abstract: Transform a string or number so that it can be stored safely in the database.
  Description: Places "' around the string and escapes the contents. As such, possible SQL injections are prevented, which is unfortunately a common security problem of a lot of web application.
  Parameter (1) `$value`: Value to be quoted and escaped.
  Result: String which can be used directly in an update or insert statement without manualy adding "' around the string.
  Example:

```
db_string("a' test") == "'a\' test'"
```

- **function insert**
  `insert($table, $fields = array())`
  Abstract: Add a new data record to a table of the database.
  Parameter (1) `$table`: String with name of table to insert the dataset into.
  Parameter (2) `$fields`: Associative array with the field names and their contents.
  Result: Boolean. True, if and only if the insertion was successful.
  Example:

```
insert('test', array('field1'=>'value1', 'fieldname2',=>'value2'));
```

- **function insert_byref**
  `insert_byref($table, &$fields )`
  Abstract: Low memory consuming function to add big datasets to the database.
  Description: If a big dataset has to be stored to the database, the memory used for storing the variables between function calls can become very huge. This function does not call any other than native PHP functions. The parameters for the data to be stored are taken as reference which also reduces the memory consumption. After this function was executed, the data to be stored is modified and no longer

usable. The code used in functions insert, query, db_string, and db_addslashes has been duplicated here.
Parameter (1) `$table`: String with name of table to insert the dataset into.
Parameter (2) `&$fields`: Associative array with the field names and their contents.
Result: Boolean indicating whether the insert was successful.

- **function update**
  `update($table, $fields = array(), $primarykey)`
  Abstract: Update a dataset in a table.
  Description: This function can be used to updated a row of a table. The primary key must be included in the data to be updated (the primary key will not be changed. The primary key is only needed to identify the row to update).
  Parameter (1) `$table`: String holding on to the name of the table to update some fields for.
  Parameter (2) `$fields`: Associative array with the field names and their contents. If the field name is prefix by '__', the value of the update is taken as is and will not be considered to be a string. Such it is possible to use SQL statements to manipulate the data.
  Parameter (3) `$primarykey`: String representing the primary key of the table whose records are updated.
  Result: Boolean indicating whether the update succeeded.
  Example:
  `update('test', array('field1'=>'value1', 'fieldname2',=>'value2'), 'pk');`

- **function delete**
  `delete($table, $conditions)`
  Abstract: Delete a row from a table.
  Description: All rows which match the conditions in table $table will be deleted. No row will be deleted if no condition is set.
  Parameter (1) `$table`: String with name of table where the rows should be deleted.
  Parameter (2) `$conditions`: String containing the condition which the row must meet in order to be deleted. The conditions are joined to a WHERE clause with function Conditions2SQL().
  Result: Boolean indicating wheter the deletion was successful.

- **function free**
  `free()`
  Abstract: Discard the query result and free the memory.

- **function next_record**
  `next_record()`
  Abstract: Retrieve the next record or rather row from a query.

Description: This function can be used to iteratively and sequentially retrieve the results of a query.
Result: Boolean. True, if and only if a next record is available.

- **function seek**
  `seek($pos)`
  Abstract: Go to a special record or row in the active query result.
  Parameter (1) `$pos`: Location of the next record to retrieve.

- **function Conditions2SQL**
  `Conditions2SQL($origtable, $conditions)`
  Abstract: Constructs from the multidimensional array $conditions a WHERE clause for an SQL query.
  Description: Prior to creation of this the function, it was possible to automatically join the tables appearing in the conditions. However,there have been some problems so the joins must be done manually here.
  Parameter (1) `$origtable`: String containing the name of the table the query mainly should run on
  Parameter (2) `$conditions`: Array with the combination of database and field names and the condition the fields must match. The database field names must consit of a table name and field name joined with a dot. If the database field name starts with a '_', the condition will not be checked and converted by function sql_condition but will be taken as is.
  Result: String with the WHERE clause.
  Example:

```
$this->Conditions2SQL('test', array('test.f1'=>'foo',
'_test1.f2' => '%2 = 0' ) ==
"WHERE test.f1 = 'foo' AND test.f2 %2 = 0
```

- **function sql_condition**
  `sql_condition($key, $value)`
  Abstract: Evaluate a regular expression pattern and generate a WHERE clause.
  Description: The text is scanned and depending on its structure it is decided to interpret it as either a verbatim POSIX regular expression, POSIX regular expression, shell pattern, ANSI-SQL LIKE pattern, range expression, comparison, or a fixed strings. The according WHERE clause of an SQL statement is then generated.
  Parameter (1) `$key`: String with the database field name.
  Parameter (2) `$value`: Value to check and transform.
  Result: String that can be used in an SQL condition.
  Example:

```
sql_condition('test', 'foo') == "test = 'foo'"
```

```
sql_condition('table.field', '2...3') ==
"table.field BETWEEN '2' and '3'"
```

- **function transaction_begin**
  `transaction_begin()`
  Abstract: Start a database transaction.
  Description: As there can only be one transaction in progress at a time, the transactions are managed by the global database object. If a second start of a transaction is requested the transaction counter in the global object is increased.
  Result: Boolean indicating whether the transaction has been started successfuly.
  Example:
  `$dbobj->transaction_begin();`

- **function transaction_commit**
  `transaction_commit()`
  Abstract: Commit a transaction.
  Description: Commits a started transaction. This is done via the global database object. Only if the global transaction count is 1 or less a real commit is done.
  Result: Boolean indicating whether the transaction has been commited successfuly.
  Example:
  `$dbobj->transaction_commit();`

- **function transaction_abort**
  `transaction_abort()`
  Abstract: Abort all transactions in progress.
  Description: Aborts all transactions in progress. This is done via the global database object and the transaction counter is set to 0.
  Result: Boolean indicating whether the transaction has been aborted successfully.
  Example:
  `$dbobj->transaction_abort();`

- **function lock**
  `lock($table, $mode = 'write')`
  Abstract: Lock a table in the database.
  Parameter (1) `$table`: String with table name or array of string with table names that are supposed to be locked.
  Parameter (2) `$mode`: String representing the mode of the lock. Only mode 'write' is supported.
  Result: boolean indicating whether the lock was successful.

- **function unlock**
  `unlock()`
  Abstract: Release all requested table locks of the database.
  Result: Boolean indicating whether the locks could be removed.

- **function affected_rows**
  `affected_rows()`
  Abstract: Get the number of rows an update or delete statement has affected.
  Result: Number of affected rows.

- **function num_rows**
  `num_rows()`
  Abstract: Get the number of rows which a select statement returned.
  Result: Number of rows.

- **function num_fields**
  `num_fields()`
  Abstract: Get the number of fields which a select statement returned.
  Result: Number of fields.

- **function nf**
  `nf()`
  Abstract: This is a shortcut for function num_fields().

- **function f**
  `f($Name,$strip_slashes = '')`
  Abstract: Get the value of a field with $Name of the current row of the results of a SELECT query.
  Parameter (1) `$Name`: String with name of the field to retrieve.
  Result: The value of the field being retrieved. If the field does not exist an empty string is returned.

- **function halt**
  `halt($msg, $line = '', $file = '')`
  Abstract: If a database error occurred, this function decides what to do next.
  Description: Depending on the contents of variable $this->Halt_On_Error the execution is stopped, a warning is printed or the error is silently ignored. Typically, all errors are ignored in a productive environment. This function is for private use of the database class only.

Class html

Abstract: This class is used to genereate HTML tags for the web based user intzerface of the testbed.
Description: This class is used to generate HTML tags like enumerations, selection lists or hidden fields from given PHP data structures of the testbed automatically.
File: common/inc/class.html.inc.php

- **function enumeration**
  `enumeration($arr)`
  Abstract: Create a list whose layout is conform to the TUD (Technical University of Darmstadt) corporate identity layout.
  Parameter (1) `$arr`: One-dimensional array with the elements to be listed.
  Result: String with the list as HTML code.

- **function Options**
  `Options($entries, $selected = 0)`
  Abstract: Creates an HTML options list.
  Parameter (1) `$entries`: One-dimensional array of strings containing the list entries descriptions indexed by the keys or names of the entries.
  Parameter (2) `$selected`: Value or array of values that should be marked as preselected (by default non is preselected).
  Result: String with the options in HTML code.

- **function Select**
  `Select($name, $entries, $selected = 0, $multiple = 0, $SubmitOnChange = 0)`
  Abstract: Create an HTML selection list for a FORM-element.
  Parameter (1) `$name`: String with the name of the selection list.
  Parameter (2) `$entries`: One-dimensional array with strings containing the descriptions indexed by a key for each that represents the element.
  Parameter (3) `$selected`: Value or array of values that should be marked as preselected (by default non is preselected).
  Parameter (4) `$multiple (default=0)`: Is the user allowed to select multiple entries?
  Parameter (5) `$SubmitOnChange (default=0)`: A submit of the FORM is triggered after the selection of the list has been changed.
  Result: String with the selection list in HTML code.

- **function hiddenfields**
  `hiddenfields ($arr)`
  Abstract: Create hidden HTML fields in the form of FORM elements from the given array.
  Description: Hidden fields can be used to transfer data within an application among several pages, e.g. when a configuration is configured over three pages.
  Parameter (1) `$arr`: Array with strings containing the value of the hidden fields indexed by the name of the hidden field.
  Result: String with the HTML Code.

Class Template

Abstract: This class is used to generate HTML pages from templates.
Description: This class centralizes any web page creation on basis of templates. The usage of such a central class has the advantage that the layout and representation can easily be changed without touching the source code of the testbed. Many functions of this class are concerned with the management of place holders. These are referred to as variables also.
Discussion: This class is originally part of the PHP library PHPlib and then has also been used in the phpGroupWare and now in the testbed.
File: common/inc/class.Template.inc.php

- **function Template**
  `Template($root = '.', $unknowns = 'remove')`
  Abstract: Constructor for the class.
  Parameter (1) `$root`: Root template directory.
  Parameter (2) `$unknowns`: String containing an order how to handle unknown place holders.

- **function set_root**
  `set_root($root)`
  Abstract: Set a new root directory where template files can be found.
  Parameter (1) `$root`: New template root directory.

- **function set_unknowns**
  `set_unknowns($unknowns = 'keep')`
  Abstract: How are unused template place holders supposed to be treated?
  Description: Any unknown place holders can either be removed ('remove'), changed to HTML comments ('comment') or keept ('keep') unchanged (compare to function finish).
  Parameter (1) `$unknowns`: String with order, either 'keep', 'remove', or 'comment'.

- **function set_file**
  `set_file($handle, $filename = '')`
  Abstract: Assign template files and define handles to them.
  Description: This function assigns handles to template files basically defining the currently active template. The file is searched for in the path which was set with function set_root.
  Parameter (1) `$handle`: String with symbolic name or rather handle of the template.
  Parameter (2) `$filename`: String containing the name of the file which contains the template HTML code.

Discussion: Multiple handles can be set by using an array as argument for argument $handle. The array has to have format 'handle' => 'filename'.
Example:
```
$t->set_file(array('f1'=>'f1.tpl', 'f2'=>'f2.html'));
$t->set_file('f3','f3.tpl');
```

- **function set_block**
  `set_block($parent, $handle, $name = '')`
  Abstract: Set a block of a template.
  Description: Extract from template with handle $parent the part that is indicated by brackets <!–BEGIN $handle–> and <!–END $handle–> and replace it with place holder $name.
  Parameter (1) `$parent`: String representing the template handle from which the block can be extracted.
  Parameter (2) `$handle`: String representing the handle for the newly created block.
  Parameter (3) `$name`: String with name of the variable which contents should replace the block. If this parameter is omitted the name contained in variable $handle is used.
  Example:
  ```
  $t->set_block('f1', 'row', 'rows');
  ```

- **function set_var**
  `set_var($varname, $value = '')`
  Abstract: Set a value for place holder $varname so it will later be replaced with the content of $value in the template.
  Parameter (1) `$varname`: String with name of place holder that later should be replaced or rather filled.
  Parameter (2) `$value`: Contents of the place holder in HTM code.
  Discussion: Multiple place holders can be set if argument $varname is an array with the contents indexed by the place holder names.
  Example:
  ```
  $t->set_var( array('name' => "$foo", 'link'=>'http://...') );
  ```

- **function subst**
  `subst($handle)`
  Abstract: Replace all place holders in the template indicated by argument $handle.
  Parameter (1) `$handle`: String representing handle of template where place holders are to be substituted.
  Result: String with the contents of the template and the replaced place holders.

- **function psubst**
  `psubst($handle)`
  Abstract: Short form for: print $t->subst($handle).

- **function parse**
  `parse($target, $handle, $append = false)`
  Abstract: Parse the contents of template indicated by $handle and store the result into a new variable $target.
  Parameter (1) `$target`: String representing the handle of the variable to store the result.
  Parameter (2) `$handle`: String representing the handle of the template that is to be parsed.
  Parameter (3) `$append`: Append `$handle`: to `$target`: finally!
  Result: String with the parse result possibly appended by $target.

- **function pparse**
  `pparse($target, $handle, $append = false)`
  Abstract: Short version for: print $t->parse(...).

- **function get_vars**
  `get_vars()`
  Abstract: Return an array with all known place holders and their contents.
  Result: Array with the contents of the known place holders of the current template.

- **function get_var**
  `get_var($varname)`
  Abstract: Return the contents of place holder.
  Parameter (1) `$varname`: String containing the name of the place holder whose contents is supposed to be returned. Use an array to get more place holder's values..
  Result: Contents of place holder $varname or an array with the place holder names as keys and their contents as values.

- **function get_undefined**
  `get_undefined($handle)`
  Abstract: Return place holders that exists in the template but have not yet been set by functions set_var or parse.
  Parameter (1) `$handle`: String representing the handle of the template to check for undefined place holders.
  Result: Array with the undefined place holder identifiers.

- **function finish**
  `finish($str)`
  Abstract: Depending of the object state 'unknown' which represents the mode for dealing with unknown place holders, unknown place holders inside string $str are changed accordingly.
  Description: Depending on the mode in object variable 'unknown', the place holder

found in string $str will be replaced, left as are, or replaced with an HTML comment (compare to function set_unknowns).
Parameter (1) `$str`: String to replace the unknown place holders in.
Result: String with the replaced unknown place holders.

- **function fp**
  `fp($target, $handle, $append = False)`
  Abstract: This is short cut for function call finish(parse(...))
  Parameter (1) `$target`: String representing the new variable storing the finished parse effort.
  Parameter (2) `$handle`: String representing the handle of the template to parse.
  Parameter (3) `$append`: Append `$handle`: to `$target`: ?
  Result: String with finished parse result.

- **function pfp**
  `pfp($target, $handle, $append = False)`
  Abstract: This is a shortcut for function call print finish(parse(...)).
  Discussion: See function fp or parse.

- **function p**
  `p($varname)`
  Abstract: Print the finished contents of a variable containing the finished parse result (compare to functions finish and parse).
  Parameter (1) `$varname`: String with name of the variable to print.

5.2.10 Example

After the preceeding description of the functions of class `template` that are concerned with templates, a short example will demonstrate how to use templates with the help of this class. It may take a long time to understand how to work with the templates, because the documentation is very confusing and the example templates used in the phpGroupWare documentation have been used different. This example hopefully gives a good start. It will be shown how to fill a table with rows. The template file looks like the following:

```
<div class="error">{errors}</div>
{search_filter}
<table>
  <tr>
     <th>{lang_name}</th>
     <th>{lang_description}</th>
     <th>{lang_actions}</th>
  </tr>
```

```
<!-- BEGIN test_row -->
  <tr bgcolor="{tr_color}">
     <td>{name}</td>
     <td>{description}</td>
     <td>{actions}</td>
  </tr>
<!-- END test_row -->
  <tr>
     <td colspan="3">
        <table>
           <tr>
              <td align="left">
                 <form method="POST" action="{url_newx}">
                    <input type="submit" name="_newx" value="{lang_newx}">
                 </form>
              </td>
              <td align="center" width="100%"> </td>
              <td align="right">
                 <form method="POST">
                    <input type="submit" name="_ok" value="{lang_done}">
                 </form>
              </td>
           </tr>
        </table>
     </td>
  </tr>
</table>
```

First, a template object is created equipped with path information where the template files can be found. This is done with:

```
$this->t = CreateObject('common.Template',
  '<path to template directory>');
```

The next step is to load the template file and define the blocks to use with

```
$this->t->set_file(array('test' => 'test.tpl'));
$this->t->set_block('test', 'test_row', 'rows');
```

After having done this, some test rows are filled with data as can be seen in the next code example. Array `$data` contains arrays as elements, one for each row. These element array contain fields named `name` and `description` that store the name and a description of the row elements, respectively. The keys of these arrays correspond to the place holders used in the template, namely, `{name}` and `{description}` but without the brackets around them. This information typically is retrieved from a storage service object (so-service object) with function `GetList()`. For this example the data is defined explicitly.

```
$data = array(
    array('name' => 'row1', 'description' => 'd1'),
    array('name' => 'row2', 'description' => 'd2'),
    array('name' => 'row3', 'description' => 'd3'),
);
foreach($data as $elem) {
    // Set name and description.
    $this->t->set_var($elem);
    // Set a link to a detailed view of that element.
    $this->t->set_var('actions',
        $GLOBALS['ui']->imagelink('common', 'details.png', '/index.php',
        '_menuaction=app.object.function&element='.$elem['name']));
    // Change the background color of the current row.
    $GLOBALS['testbed']->nextmatches->template_alternate_row_color($this->t);
    // Any data for the current row is set, now append the data of this row to
    // the previously generated rows.
    $this->t->fp('rows', 'test_row', true);
}
```

Finally the rest of the the remaining place holders of the template are filled:

```
$this->t->set_var('lang_name', lang('Name'));
$this->t->set_var('lang_description', lang('Description'));
$this->t->set_var('lang_actions', lang('Actions'));
$this->t->set_var('lang_newx', lang('new X'));
$this->t->set_var('lang_done', lang('Done'));
$this->t->set_var('url_newx', $GLOBALS['ui']->link('/index.php',
                  '_menuaction=app.object.Edit'));
```

The completed template is printed to the screen in the form of an HTML page with:

```
$this->t->pfp('out', 'test');
```

5.2.11 Notes

Since the testbed in its first stage of development was implemented as a single user system, bo-services have not been used to access the data. Instead, so-services are used directly. This has not been changed yet, but should be changed in future versions of the testbed. The dependencies between the different kinds of service objects are shown in figure 5.2 on page 233.

It is possible to have different classes which represent the same functionality. In phpGroupWare this is used for example for user administration. That way it is possible to retrieve and store user information in a database, get the information from a LDAP[4], or

[4]Lightweight Directory Access Protocol

to implement something individually. Any such option will then be implemented by mutually replaceable classes. All those classes will have the same interfaces and functions, they simply handle the retrieval and storage in different ways.

The testbed is implemented in an object oriented style. However, this style is not strictly object oriented. Instead, the usage of objects to represent the various parts of the testbed was primarily chosen to provide recurring services like storage of data structures or presentation of data to the user within an unified interface. As a result, the implementation does not fully follow the typical object oriented approach. The objects implementing recurring services communicate via complex data structure which is typically not found in a strict object oriented approach. PHP objects are primarily used to encapsulate and hide function names from the global PHP name space[5] in order to keep status information inside these objects. For example, when storing a problem instance, no problem instance object itself is stored. Instead, a complex data structure representing the problem instance is given to a storage object which represents the storage service for problem instances. This service object now handles the storage of the data structure, i.e. of the problem instance. If an error occurred while storing the problem instance, the service object returns a false to the original principal. The principal then can retrieve more information about the error that occurred from the service object. All data structures used are listed in appendix A.2 on page 289. They are modeled within PHP as multidimensional arrays, indexed by keys.

[5]Normally, a function name can only be used once, except the function name is used and encapsulated inside a PHP class

5.3 Extending the Testbed

The testbed is designed to be easily extensible. The modular structure of the testbed with different applications and different groups of service classes (see subsection 5.2.1 on page 232) together with the featured template functionality gives developers powerful tools to extend the testbed. Additional functionality, beside the core or global functions, should be put into separate new applications. Only if the extension will affect or will be used by all applications, new functionality should be put in the 'common' application. Recall that in directory 'common' all basic classes used for accessing the database, generating templates, accessing problem types, navigating through lists, and icons commonly used can be found. The directory structure for new applications must follow the directory structure as described in subsections 5.2.3 and 5.2.5 on pages 235 and 239.

In the following, some guidelines to extend the testbed are given. A good way to start extending the testbed is to look at the source code of the existing applications.

5.3.1 Using FORMs in UI

The testbed framework provides an easy interface to extract all information from an HTML page form via function `ExtracFormVars()`. The information can be user input entered recently or some data that were stored temporarily in order to convey it between two functions subsequently preparing the same web page. To use this function correctly, the variables inside an HTML form to be processed must follow the name scheme described next:

- All input buttons must start with a '_'. All variables starting with a '_' are not extracted via `ExtracFormVars()`. Yet, they still are accessible via function `get_var` or via variable `$GLOBALS['HTTP_POST_VARS']`.

- All variables not starting with a '_' are extracted. The '_' inside the variable names are used to split the name into an array structure. Such it is possible to retrieve complex data structures from HTML pages directly. The data structures retrieved can then be passed directly to the so- or bo-service objects. There is no need to convert or build these data structures in a ui-service object itself.

- Temporary or help variables inside the HTML form should also start with a '_'.

All applications use this naming convention for retrieving information from web pages. A good example is file *class.uialgorithms.inc.php* in directory `TESTBED_ROOT`*/algorithms/inc/*.

5.3.2 Extending the Search Mask

The search mask, i.e. the page of the 'Search Filters' submenu can be changed quite easily. This is helpful, if during the development of special search filters the input fields for parameters are not enough and more input fields are required. The overall number of parameter input fields allowed per object type can be specified in file *config.php* located in directory `TESTBED_ROOT/` with line

```
define('SEARCH_LISTS',5);
```

The number indicates the maximum number of parameter input field for each object type that feature parameters, i.e. jobs, configurations, and algorithms. Which kind of parameter input field will be shown can be set in file *search.tpl* located in directory `TESTBED_ROOT`*common/templates/default/*. This file contains the template for building the search mask page including the entries for the parameter input fields. Depending on the number of parameter input fields that were set in file *config.php*, the following lines can be added at the appropriate place[6]. One line for each additional parameter input field desired is needed. Place holder `<objectType>` has to be replaced by either `job`, `configuration`, or `algorithm`. Place holder `<No>` has to be replaced by a number ranging from `1` until the maximum number of parameter input fields per objects type as defined in file *config.php*. Note that no duplicates are allowed. The first example line will create a parameter input field with a selection box used to request the parameter name combined with a text input field for the corresponding parameter value. The second example will create for both parameter name and value a text input field.

```
<td>{sel_<objectType>param_<No>}</td><td>{input_<objectType>value_<No>}</td>
<td>{input_<objectType>param_<No>}</td><td>{input_<objectType>value_<No>}</td>
```

Note that when changing file `config.php`, it must be changed directly. It is not possible to change a copy and overwrite the old version. This will yield an error. See section 4.6 on page 217 for more information.

[6]New entries for the input fields have to be placed in an HTML-table! See the original placement in file *search.tpl*

6 Future Work

The testbed described in this document is designed to ease the work for experimenters and to help them to concentrate on the development of algorithms instead of evaluation and management scripts. The aim of the testbed is to be spread and to be useful for scientists experimenting with algorithms by enabling them to share their results and make their work more transparent and reproducible for each other. During the development and usage of this testbed, a lot of possible new features and extentions have been identified. Unfortunately, there has not been enough time to implement and document these extensions yet. This testbed is open source, i.e. anybody who wants to can extend the testbed under the policy of the GNU General Public License (*http://www.gnu.org/licenses/licenses.html*). Further information about version control and coordination of development effort can be found on the testbed home page [62].

This chapter is devoted to future work for the testbed. It is comprised of a loose list of extensions to the testbed that is intended to give guidelines of the directions of further development of the testbed. The order of the individual items is not according to importance.

Meta Data At the moment, additional information associated to objects of the testbed except those attributes already implemented can not be added. Adding new static attributes encompasses changing the structure of the database and the web front end. It is desirable to give user means to add additional information dynamically to objects. For example, problem instances can then be assigned an optimum solution (value) by a user. As well, it may be possible to store a maximum runtime for an algorithm to run on a specific problem instance. This value could be used automatically, if no other value is set for a maximum execution time of jobs. Sometimes it is preferable to make the runtime of algorithms dependent not only of the instance size (see the implementation of the testbed's dummy module for how to achieve this as described in subsection 3.2.1 on page 74) but also to make the runtime dependent on individual problem instances. Instances, even of the same size, can be quite differently hard to solve. In order to obtain reasonable good results, algorithms should be run longer on harder instances. At the moment, the testbed does not provide any means to attach a recommended runtime to a problem instance.

Flow of Data Another kind of meta data is concerned with the flow of data between the sequence of modules an algorithm consists of. Algorithms consist of modules ordered sequentially. Accordingly, the data flows sequentially through the modules of an algorithm when it is executed. This flow of data is realized via temporary files. Each module reads its input data from a temporary file as assigned by the testbed via the `--input` flag. Each module writes its output to a temporary file as assigned by the testbed via the `--output` flag. The testbed takes care that the values of these flags match the respective output and input files to transfer the data form one module's output to the next module's input.

Currently, only one flow of data between modules of an algorithm is supported, since only one single temporary file (compare with subsection 2.3.1 on page 14) can be realized by the mechanism described just now. If a module, for example a module implementing a learning algorithm, produces two or more output files or requires two or more input files, a wrapper must be provided to encode two or more files into one. Not only modules requiring two or more input or output files will need such a wrapper but also modules preceeding or succeeding such modules will need a wrapper to be able to extract the parts they are interested in and to ignore the rest. Additionally, if more than one separate parts have to be conveyed via one single file as described just now and some parts simply have to bypass some module of an algorithm's sequence, all bypassed modules will require a wrapper that implements the bypass, too, for the same reasons. Altogether, this procedures is feasible, but cumbersome.

The logic consequence is to extend the testbed to allow multiple input and output files for each module. The user must be enabled to model the flow of data, i.e. to model which output of which module serves as input for which other module, when creating an algorithm via the user interface. In particular, by allowing bypasses, this way the user can flexibly define multiple flows of data. Finally, to go one step further, the restriction that algorithm might only exist of a linear sequence of modules can be dropped in principle, too, by allowing any kind of acyclic net of modules to form an algorithm not just linear ones.

New Dynamic Object Types To seize and elaborate the suggestion of the preceeding paragraph 'Flow of Data', the testbed should be able to incorporate any kind of input files, not just problem instances. For example, algorithms employing a machine learning approach typically either need to read in the definition of a learned function or need to output such a definition or both. Planning algorithm split any posed problem in to parts instantiated by two separate files, too: One file contains the general description of a domain and which actions are possible and which effects they have, the other file then encodes a particular task in the form of a starting state and a desired goal state. In the case of a learned function, the contents might vary from one run of an algorithm to another in contrast to problem instances that remain constant. Altogether, it seems

desirable to enable a dynamic integration of new types of data objects for use as input and output of modules that possibly change over time. i.e. has to be updated in the database.

Database Driven Events and Triggers One possible extension to the testbed is to add an event and trigger based subsystem. That way, it is possible to send an email when all jobs of an experiment have finished or when a job failed. At the moment, a similar mechanism does exists in the form of hooks (see subsection 5.2.5 on page 239), but it is better to trigger actions such as sending an email via the database directly.

Integration of Problem Instance Generators The integration of problem instance generators is appreciable. The user can define the parameters to generate a problem instance which then is automatically stored in the database. At the moment, problem instance generators can be integrated in the form of modules preceeding the module actually representing the algorithm. However, the problem instances generated such can not be stored in the database automatically.

Multi User Operation It certainly is desirable to allow more than one user to work on the same database. Currently, this is possible in a basic version only with some limited access control and protection of other user's data, if two or more user work on the same database. Right now, each user employs its own separate database in a multi-user setup. That way, no user can destroy other user data, if the database access is restricted, but exchanging or sharing data is only able via im- and export facilities which is cumbersome. Testbed database internal access control is implemented for categories only. Since the concept of categories was taken from phpGroupWare the categories have support for multi user operation. The rest of the testbed, however, lack multi user support with access control. This support can be integrated without having to change the testbed structure, since with the help of phpGroupWare access control is handled by bo- and so-classes (see subsection 5.2.1 on page 232).

Using the Testbed via Internet The user interface of the testbed is web based. Therefore, it is no problem to use the testbed via Internet. As was mentioned before, however, using the testbed from remote computers comes with some complications. The most important aspect is access control. At the moment, the usage of the testbed from remote is fairly restricted, since basically no substantial right on the testbed local machine can be granted to remote user. This restricts the application of the testbed via Internet from remote. In order to make the testbed work properly via Internet, extensive access control has to be established such that the testbed can not be abused. This has not been implemented fully yet. For example, it must not be allowed for a user to accidently or

on purpose fill the database and hence the hard disk with junk data by running millions of wrong or faked algorithms.

Registering a Module via the Web Front End If a module were to be registered via the web front end of the testbed, the web server would have to change its user ID to the ID of the user that wants to register the module, because the web server normally does not have write access to the user directories. The apache web server has a mechanism to do this with the suEXEC wrapper. For more information about suexec see the online documentation 'Apache suexec Support' on the HTTP server project's Web site at *http://httpd.apache.org/docs/suexec.html* [76]. This also requires authentication of the user himself, so that any modifications of the user directories can only be done by the user itself. This extension proposal is closely related to the two previous ones.

Parameter Subrange Checking When generating a module definition file automatically, currently, only the type information and some restricted intervals in case of number are translated into regular expressions in order to check any user input setting parameters based upon the information given by a module in the form of its command line interface definition output. The reason is that the definition of arbitrary intervals of the form $\#_1 a, b \#_2$ with $a, b \in \text{REAL} \cup \{\infty\}$, $\#_1 \in \{\,(\,, [\,\}$, $\#_2 \in \{\,)\,,]\,\}$ in terms of a regular expression is not easy and may produce huge regular expressions. This caveat could be detoured by checking parameter values set by the user through PHP directly instead of a textual check by means of regular expression checking. The current makeshift is to provide two external Perl [65, 66] programs that automatically produce an appropriate regular expression which then can be used for checking user input setting parameter values (see section 4.2 on page 181 for how to integrate its own regular expressions for subrange checking into module definition files). The Perl programs for regular expression generation can be found in directory `DOC_DIR`*/scripts/utility*. The files are named *gen_subrange_a_b.pl* and *gen_subrange_a.pl*. The former can be used to specify intervals of type (a, b), $(a, b]$, $[a, b)$, or $[a, b]$ with a and b being real numbers. The latter can be used to specify half-open intervals of the form $\{b \in \text{REAL} \mid b \# a\}$ with $\# \in \{<, \leq, >, \geq\}$ and $a \in \text{REAL}$. The usage of the programs is explained by calling the programs without any parameters. Note that the output regular expression is in Perl notation and that the programs come without any warranty: They have not been tested extensively.

Calculation of the Computation Time of an Experiment In principle, the maximum computation time of an experiment can be computed by the testbed. This computation time could then be presented to the user when an experiment is created, to give the user an impression of the temporal scale of the experiment. Since all information about the number of tries and the maximum computation time for a try are known for each module, the theoretical net runtime could be computed.

Multi language support At the moment, the testbed is only available in English. Some parts are already capable of supporting multiple languages, but this support is still missing completely in some other parts, while in yet other parts simply the translations are missing. Functions to translate are already implemented in phpGroupWare and can easily be included into the testbed.

Experimentation Specification Language Current tools for supporting experiments with algorithm including this testbed are only concerned with data management, execution control, and statistical evaluation of experiments as separated and relatively independent processes. In particular, no feedback loop from statistical evaluation results back to the specification of subsequent parts of an experiment is provided. Sometimes, for example, the further course of an experiment is dependent on some preliminary experiments. In fact, main experiments and preliminary experiments are strongly connected and are really one coherent experiment that should be specified and run together. Ideally, the actual specification of the main experiment can be chosen automatically from among a set of alternatives in dependence of the outcome of the results of the preliminary experiments. So far, an experimenter sets up dependent experiments one after another manually. By providing a modeling mechanism for these dependencies, the process of experimentation can be automated further. For example, if one is about to tune an algorithm, the following procedure for automatic tuning could be specified and implemented with such an experiment specification language: Start with a set of all promising fixed parameter settings and run these on a number of problem instances. Do statistical tests and discard all fixed parameter settings that are significantly worse than the best fixed parameters setting found during these runs. Repeat this procedure until only a small subset of fixed parameter settings has survived. This procedure was implemented by hand in [4, 5], but could have been implemented with an experiment specification language as envisioned here.

Allowing the user with the help of some language constructs to specify all details of an experiment would give experimenters new degrees of freedom for automating the process of experimentation. Details of experiment specification comprise starting an experiment with given values, evaluating the results, and, depending on these results, starting different possible subsequent experiments until a certain abortion criteria is reached. In essence, providing an experiment specification language with such constructs would elevate the specification of experiments to a higher level allowing for even more reproducibility and reusability. Experiment specifications including feedback loops from experiment results to automatically alter the further course of an experiment depending on earlier results can be viewed as a kind of template describing how to conduct experiments. By including some mechanisms into the specification language to make these templates more generic, they can be reused quite efficiently. In effect, an experimentation specification language then is nothing else than a programming language that can control all important aspects of conducting experiments. By designing such a specifica-

tion language properly and carefully, experiment specifications could be described more or less declaratively, thus making experiment specification human readable. As is the case with the commonly used and agreed upon pseudo code for describing algorithms, a declarative experiment specification language can be used to precisely communicate experiment settings. In principle, having accumulated a lot of template experiments encoding the expert knowledge about how to properly conduct experiments including the proper analysis of results, experimenters examining algorithms need not know and worry too much about statistics and topics such as experimental design to perform scientifically sound experiments: They just chose the proper experiment template, plug in their algorithms and check the results produced. The specification language envisioned here can be easily used to specify any procedure of the field of experimental design [17, 18]and hence a lot of already existing and working procedures for testing algorithms could easily be automated, too.

In principle, it is feasible at the moment to include feedback loops into the testbed by extending the PHP code of some parts. However, the programming language for flexible experiment control introduced such in the form of PHP and the existing implementation of the testbed is too complex and not readable for non programmers. Additionally, no management for such extensions is provided. However, since the testbed already provides for a declarative specification of experiments and provides means for extracting and analyzing experimental results by means of scripts, this infrastructure could be used to superimpose an experiment specification language.

Automatic Detection and Recovery of Crashed jobs Up to now, a crash of a job is only detected if this job exceeds the maximum runtime allowed for jobs as described in section 3.4 on page 136 on page 140. If the maximum runtime allowed is to low, some jobs might not be allowed to finish at all, even if they could. It is desirable to provide a more elaborate detection of jobs not running properly, e.g. by periodically checking the jobs currently running. Lost jobs can be restarted anew already, but it would be desirable, if the testbed could detect at which point of execution a job aborted. This presupposes provision of continued information about the status quo on the part of a job. Provided a job can give this information, the job could be continued automatically at the point it aborted, possibly saving substantial runtime.

Therefore, it might be useful to provide some code fragments that can be incorporated into module implementations which provide information about the course and current state of execution of jobs and which help in supervising jobs during execution perhaps by being able to answer to signals. Possibly, these code fragments can be placed in the wrappers of modules.

Enhancing User Friendliness The testbed is designed to ease the work of experimenters. One part of this support is to provide a user interface that efficiently supports

this work. Therefore, the user interface of the testbed has incorporated quite a lot of usable actions for the most frequently recurring tasks such as searching for and grouping of data, deletion, editing and copying of data, and so on. In what follows, a list of further possible user interface enhancements is presented:

- All columns in all submenus should be eligible for ordering submenu entries.
- Incorporation of new columns for certain submenus that give links to important components of entries. That is, important components of entries in submenus, for example the algorithm of a configuration, are named *and* hyperlinked. These links then not only present the name or a short description of an entry's component, but can also be clicked, opening a new page with a detailed view of the specific component. The same mechanism could be implemented for the detailed view of objects in the testbed. Important components for some types of object that are potential candidates for the linking mechanism are given in the following list. The hyperlinked components are given after the colon.
  - Configuration: Algorithm.
  - Job: Experiment, configuration, algorithm, problem instance.
  - Experiment: Jobs, problem instances, configurations.
  - Algorithm: Module.

Categories Categories are the most important means to organize data within the testbed. Categories can be compared to directories in a typical hierarchical file system. Both group coherent data together and thus enable quick retrieval of this data. However, as implemented in the testbed, categories are far more powerful. Dynamic categories for example are updated automatically, i.e. new objects that fulfill the requirements specified by a dynamic category's SQL statement will be added automatically. The user does not need to interact, i.e. to clean up the object space manually by distributing new objects into the appropriate directories. If the arrangement has to be changed, only the category specification in the form of an SQL statement has to be changed. No manual moving of objects is necessary as would be the case in fixed directories organized hierarchically. This is possible mainly because one object can be present in multiple categories. This is not possible with the help of directories without having to copy it or without having to create links which result in some semantic difficulties. Besides, static categories can mimic the typical file system grouping behavior, even grouping different types of objects together.

However, categories are not yet implemented for all objects in the testbed. It is desirable to introduce categories for data extraction and analysis scripts, too, as well as for categories themselves. This implies that categories and scripts will be included into

the search filter search mask, too. Unfortunately, this needs some major changes in the testbed database structure which is the reason why it has not been implemented yet.

In general, if all types of objects that are managed or involved in the testbed were subject to forming categories, categories could be used to restrict the selectable elements in selection boxes an lists throughout the testbed by just placing an additional selection box with the categories available for the object type that is to be selected via a selection box or list. This might be useful if the number of objects in the testbed increases as time goes by and accordingly the number of elements in selection boxes and lists become huge. As a first remedy for limiting the number of selectable entries in certain selection lists or boxes, the problem type could be employed to filter the total number of possible choices, e.g. for the parameter selection lists in the search filter mask (compare to next paragraph).

In order to give even more flexibility to the usage of categories as the major means of organizing data, static and dynamic categories could be married completely. That is, each category can have a dynamic part as expressed by an SQL statement and a static parts that is defined by the user manually. This should be implemented for all types of objects. Furthermore, it would be desirable to provide a separate detailed view for categories where the user can organize the static and dynamic parts of a category. Finally, an elaborate edit and copy functionality is needed for categories, too.

Extending the Search Filter Generation Tool The search mask for automatic generation of search filters can be further extended. If modules become ordinary objects that can be searched for, a new headline with module specific input fields can be added. Such it will be possible to search for all jobs whose algorithm contains a module featuring a parameter with name `xyz`, for example. This is not possible at the moment. In general, each new type of object eligible to be searched for will have to be incorporated into the search mask with its own headline, for example scripts and categories. Furthermore, handling of arbitrary numerical intervals and timestamp intervals could be enhanced. Finally, categories could be used most profitable within the search mask to confine selectable entries in selection boxes. A first filtering of selectable entries in selection boxes and lists in the search mask could be performed based on the actual problem type chosen. The default problem type basically is then be used as a category. In fact, it is a very special category of great importance in practice.

Further extensions could comprise that algorithm parameters should not only comprise hidden parameters, but all parameters that have been set to a default value and all parameters that are supported by an algorithm. This enables the user to specify search filters such as all algorithms that feature a parameter with name `xyz` or all algorithms that feature a parameter `xyz` set to default value `zyx`. Perhaps parameters can be equipped with a further internal attribute that indicates whether it was hidden, or set as internal parameter upon module definition file generation or whether it was set to a

default and which default.

Data Extraction – Ordering the Results When extracting data, the columns of the final result table that are displayed or exported can be selected using button 'Calculate fields' (compare to subsection 3.3.10 on page 125). Undesired columns can be omitted. Additionally it is desirable to have a possibility to define an order on the columns which is used to sort the rows of the table. The entries of the first column in this order are used to sort the rows of the table first, ties are broken with the next column, and so on. This is useful if any subsequent statistical evaluation, e.g. plotting of plots, is dependent on the order of the data that is to be processed. Right now, a specific order can not be guaranteed but is up to the implementation of the database employed. However, among one implementation, the order is fixed.

Database Management The management of the testbed's database can be made more transparently to the user by including a direct web based interface to the database. An interface like this does exist already as have been mentioned in section 3.4.5 on page 142. This web based user interface to PostgreSQL is called phpPgAdmin and is described in [45]. It could be installed together with the testbed per default and could be linked appropriately to the testbed user interface.

Porting the Testbed to Windows The testbed by itself is not platform dependent, since the software it is based upon (compare to subsection 3.1.2 on page 54) in principle is available on many different platforms. The testbed itself was tested on and developed for Linux, but it should also be portable to Unix systems. A port to windows, however, could be more difficult, but not impossible.

Extensive and context-sensitive online help Online help within the testbed currently is only available for the construction of regular expressions for search through submenus via a 'Help' button (see point 5 in figure 3.17 on page 100 in subsection 3.3.2 on page 100. Currently, the user manual is available in HTML format as well, but it is not specifically tailored to give online help to issues directly from within the testbed.

Using MySQL The database used for the testbed is PostgreSQL ([56]). The main reason to use PostgreSQL was its support of transactions. In newer version, the MySQL database also supports transactions. Since both databases can be addressed from PHP and the the main interaction between the testbed's GUI and its database are standard SQL commands, it should be possible without major effort, to support MySQL database with the testbed, too.

Automatic Documentation of the Experimental Environment The ExpLab [63], a tool set for computational experiments, aims in similar direction as the testbed. It is designed to 'support the running, documentation, and evaluation of computational experiments' ([64]). ExpLab concentrates on documenting the hard- and software environment of an experiment to support reproducibility and has tools for automatic extraction of data from any result similar to the data extraction scripts of the testbed. The task of specifying experiments is not covered as much as by the testbed. Specific data management of specification data and data produced by the experiments is not integrated. The documentation engine is quite elaborated and a lot of ideas and functionality from it could also be implemented in the testbed. For example, an automatic recording of the hardware and software environment an algorithm is run on can be included in the result files. This certainly helps when an experiment is supposed to be rerun in order to reproduce the results. The testbed could automatically collect data of this kind, add it to the job results, and extract it automatically, too, when needed.

Hardware Classes To act on the suggestion of the last paragraph 'Automatic Documentation of the Experimental Environment', the handling of a heterogeneous network of computers by the testbed could be improved. Currently it is possible to arrange the computers in a network that are addressable by the testbed server, i.e. which run job servers that are connected to the server, into classes of machines with equivalent computing power, so called aliases (compare to subsection 3.3.14 on page 134). It would help a lot if this classification with respect to computational power could be done automatically, for example by automatically scanning the hard- and software environment of a machine, possibly by means of benchmarking.

Currently, a user who wants to distribute jobs across a heterogeneous network of computers has the problem that machines are differently powerful. Some machine are faster than others, but experiment often need some fixed amount of runtime which has to be the same for all jobs of the experiment. To remedy this, the different machines of the network can be benchmarked and assigned a factor that identifies its speed relative compared to a standard reference machine. The user determines the runtime for its experiment with respect to the reference machine. For each job, the machine it is run on is identified together with the factor representing its relative speed with respect to the reference machine. The actual runtime of the job then is determined by multiplying the experiment runtime by the machine factor hence ensuring that the job runs approximately as many operations as it would on the standard reference machine. In practice, the user has to split its experiment into smaller ones and has to tailor them for each machine in the network by computing the relative speed factor and setting the appropriate actual runtime for the in the experiment setup manually. This is cumbersome and needs support. If each alias or equivalence class had assigned a relative factor representing its speed compared to a fixed standard reference machine automatically or manually, the relative factor could be used to alter the runtime of jobs run by the testbed transparently

for the user.

Multiple Runs One of the optional requirements of the command line interface definition format of the testbed (compare with subsection 2.3.1 on page 15) suggests to enable a module to autonomously and transparently repeat its execution in the form of several repeated tries by itself. This is useful, if the algorithm that is implemented by the module is randomized and hence several sample runs are needed to enable reliable prediction of the module's performance. A problem arises when an algorithm in fact consists of more that one module. If two or more modules provide means for autonomous repeated runs via a parameter flag, each module will repeat independent from the others. The results of all repetitions will then be put to the output file which serves as input for the next module on the sequence. If the modules are not provided with a wrapper that can extract the individual repetition parts and input one such part after another, the algorithm as whole might not work. Additionally, the next question is whether each input part is repeated several times or whether each input part is repeated exactly ones. The, however, if the number of repetitions is set differently for the individual modules, an assignment problem for the repetition of input parts arise.

Currently, it is not possible to tell the testbed to repeat an algorithm as a whole. It is appreciable to enable such a mechanism, so the testbed takes care of repeating a whole sequence of modules a number of times and summarizing the result into a proper result file automatically. Additionally, other mechanism for setting individual repetition scheme for individual modules of an algorithm which then are executed by the testbed automatically are conceivable.

Options for Parameters Consider the following case: A parameter must be called for a module, but it must not be set to a default value, i.e. it is absolutely required for the parameter that the user sets it either upon algorithm or configuration creation. Such requirements can occur for example, if there is no sensible default value for a parameter. This scenario can not be modeled by the testbed, since setting a parameter as internal parameter in a module definition file (compare to section 4.2 on page 181) will set this parameter always to a fixed default value. Additionally, it might be useful, if a user can mark a parameter as mandatory to be set when creating an algorithm or when this algorithm is configured. Incorporating such additional options for parameters into the testbed and the command line interface definition output format seems worth the effort.

Conveniences Sometimes that testbed still is difficult to handle since a lot of different parts have to be integrated which not always works completely smoothly. For example, data extraction and analysis scripts might be buggy. Up to now, however, no special support for the development of scripts has been integrated. Also, error within scripts and error messages from scripts are not yet presented coherently to the user; they are simply

output the order they occur, either issued by the PHP engines or a scripts employed. It certainly would be helpful to output any error or status messages concisely and labeled according to the source. The same way, debugging support for scripts will greatly ease script development within the testbed.

Another source of problems is the installation process. An auto-installer would be a most appreciable tool, since the emphasis of the testbed tool lies on easy usability and this incorporates installing the testbed in the first place., better development and error message support for scripts. Although the installation description in section 3.1 on page 51 is held very explicit elaborated, it can not cover all peculiarities and possible errors, mistakes and problems that occur during the installation.

A Source Code

This chapter provides some code examples illustrating the use of manually adjusted module defintion files and shell wrappers. Furthermore, some testbed internal PHP data structures are presented. The examples of this chapter are located in directory `DOC_DIR`*/examples/modules*.

A.1 Modules

This section contains examples of source code of a module definition file that adapts the execution part. The module that requires adaption is the standard example module of this user manual, namely the dummy module (see subsection 3.2.1 on page 74). The module definition file has been modified in a way that only a subset of all parameters are supported anymore (they will not show up in the testbed after registration anywhere) and the execution part has been adjusted. This adjustment assumes that the module executable writes its output to a fixed file (name). Hence, a small wrapper has to be employed to rename the output file to the name that was given as parameter by the testbed. Finally, all performance measure information is omitted.

A.1.1 Example: Pruned Dummy

```
<?php
/*
**  Description:
**  Wrapper for pruned dummy module.
*/

class module_PrunedDummy extends basemodule
{
     /*  Name of binary/executable. No need to specify a path here,
     **  if binary put in directory TESTBED_BIN_DIR/<arch>/<os>/<modulname>/.
     **  Otherwise use an absolute path starting with a /.
     **      See user manual for more information.
     */

     var $executable = 'Dummy';
```

```
/*  Description of module. See user manual for a full list of
**  featured attributes.
*/

var $ModulDescription = array(
  'module' => 'PrunedDummy',
  'problemtype' => 'Dummy',
  'description' => 'Example for a module that needs an adjusted exectuion part in \
                    it module defintion file. Example module itself is dummy module.',
);

/*  Parameter description. See user manual for a full list of
**  featured attributes.
*/

var $ParamDescription = array(
   'input' => array(
      'description'  => 'Input-file',
      'cmdline'      => '-i',
      'cmdlinelong'  => '--input',
      'typ'          => 'filename',
      'paramtype'    => 'FILENAME',
      'defaultvalue' => 'a.dat',
   ),
   'output' => array(
      'description'  => 'Output-file, \'INPUT\' being the name of the input file',
      'cmdline'      => '-o',
      'cmdlinelong'  => '--output',
      'typ'          => 'filename',
      'paramtype'    => 'FILENAME',
      'defaultvalue' => 'INPUT.out',
   ),
   'tries' => array(
      'description'  => 'Number of trials (=repetitions) of algorithm',
      'cmdline'      => '-r',
      'cmdlinelong'  => '--trials',
      'typ'          => 'int',
      'paramtype'    => 'INT',
      'condition'    => '/^[+]?[0]*[1-9][0-9]*$/',
      'paramrange'   => '>0',
      'defaultvalue' => '10',
   ),
   'minTime' => array(
      'description'  => 'Maximum time limit',
      'cmdline'      => '-t',
      'cmdlinelong'  => '--time',
      'typ'          => 'int',
      'paramtype'    => 'INT',
      'condition'    => '/^[+]?[0]*[1-9][0-9]*$/',
      'paramrange'   => '>0',
```

```
        'defaultvalue' => '1',
    ),
    'randomType' => array(
        'description'  => 'Which probability distribution to use for randomization?',
        'cmdline'      => '-e',
        'cmdlinelong'  => '--randomType',
        'typ'          => 'switch',
        'paramtype'    => 'BOOL',
        'condition'    => '/^(true|t|tt|y|yy|yes|ja|1|false|f|ff|n|nn|no|0|nein)$/i',
        'defaultvalue' => 'TRUE',
    ),

);

/*  Description of module's performance measures, if run as
**  the last one. See user manual for a full list of
**  featured attributes.
*/

var $PerformanceMeasure = array(),


/*  Run the module with the given parameters. For the parameters
**  the logical name is used and the class itself replaces the
**  logical names with the command line options
*/

function Exec( $params )
{
    $output = $params['output'];
    unset($params['output']);
    $ok = parent::Exec($params);
    if ($dir = @opendir(".")) {
        while($file = readdir($dir)) {
            if ( preg_match('°^.+\.rep$°', $file ) ) {
                $ok &= rename($file, $output);
                if (!$ok) {
                    $this->Error = "Error! Renaming of the output file failed.";
                } else {
                    $found = true;
                }
                break;
            }
        }
        if (! $found) {
            $ok = false;
            $this->Error = "Error! No corresponding output file found.";
        }
        closedir($dir);
    }
    return $ok;
```

```
    }
}
```

A.1.2 A Wrapper for an Executable

If an module executable does not comply to the minimal requirements for modules to be integrated into the testbed (see section 2.2 on page 11), a shell wrapper can be written that make up for any deficiencies by simulating a compliant interface. In the following, a shell wrapper is presented which makes a module executable compatible to the testbed. The original executable only has two parameters. The first parameter is the input file, the second parameter indicates to use some magic inside the algorithm. Additionally, the output of the result is written to standard output or rather console and the executable has no parameter that can limit the run time. For the shell wrapper to work properly, it is required that the executable is in the same directory as the wrapper and that its name is 'binary'.

```
#!/bin/bash

# Provision of help output.
helpoutput() {
echo "test module

Call:
 $( basename $0 )
 [-h|--help]
 [-i|--input]
 [-o|--output]
 [-t|--maxTime]
 [-v|--tries]
 [-x|--usemagic]

begin parameters
  -h --help     NO       Help     Get help
  -i --input    FILENAME in.dat   Input File
  -o --output   FILENAME out.dat  Output File
  -t --maxTime  INT:>0   1        Maximum CPU time to run in seconds.
  -v --tries    INT:>0   1        Number of repetitions.
  -x --usemagic BOOL     FALSE    Some special magic tricks to run faster
end parameters

begin performance measures
  best        REAL
end performance measures

" 1>&2
}

# If no parameter was entered, show the help output.
```

```
if [ $# == 0 ]
then
  helpoutput
  exit 1;
fi

# Define default parameter values.
INPUTFILE=in.dat
INPUTFILE=out.dat
USEMAGIC=0
MAXTIME=10
TRIES=1

# Parse the command line arguments.
while [ $# -gt 0 ]
do
  case "$1" in
  -h | --help )
    helpoutput
    exit 1
    ;;
  -i | --input )
    INPUTFILE="$2"
    shift
    ;;
  -o | --output )
    OUTPUTFILE="$2"
    shift
    ;;
  -t | --maxtime )
    MAXTIME="$2"
    shift
    ;;
  -v | --tries )
    TRIES="$2"
    shift
    ;;
  -x | --usemagic )
    USEMAGIC="1"
    # no shift needed, cause usemagic has no parameter
    ;;
  esac
  shift # shift the analyzed parameter
done

#
# Add some checks, whether parameters entered are useful (omitted here).
#

# Set maximum run time via ulimit.
ulimit -t ${MAXTIME}
```

```
# Try to disable the buffering of pipes to prevent loss of information
# when the executable is killed by ulimit.
ulimit -p 0

# Make MAXTRIES runs of the executable.
(
TRY=1
echo "begin problem $( basename "${INPUTFILE}" )"
while [ ${TRY} -le ${TRIES} ]
do
  echo "begin try ${TRY}"
  $( dirname $0 )/binary ${INPUTFILE} ${USEMAGIC}
  # Additionally, the output of the execuatable could be piped through a sed- or
  # awk-script to make the output conform to the testbed's standard output format.
  echo "end try ${TRY}"
  TRY=$(( TRY + 1 ))
done

# Write output to output file.
echo "end problem $( basename "${INPUTFILE}" )"
) > ${OUTPUTFILE}
```

A.2 Data Structures

In this section internal data structures in the form of PHP data structure of different service or storage objects are shown as they are returned by services. A service expects the same format when an entry is to be saved or updated. As those structures may change, the current structure can be get by calling `testbed dump <appname>.so<classname>` via the CLI. See subsection 3.4.7 on page 145 for more information)

A.2.1 Structure algorithms.soalgorithms

```
Array(
    [algorithm] => 'dfsdf'
    [problemtype] => 'QAP'
    [description] => ''
    [hiddenparams] => Array(
      [0] => 'lsmcqap_1_optimal'
      [1] => 'lsmcqap_1_time'
      [2] => 'lsmcqap_2_tabu'
      [3] => 'lsmcqap_2_time'
      [4] => 'lsmcqap_2_trials'
     )
    [modules] => Array(
      [1] => 'lsmcqap'
      [2] => 'lsmcqap'
     )
   )
```

A.2.2 Structure common.sohardware

```
Array(
    [rscript] => 'example'
    [description] => ''
    [script] => 'cat("hello world!");'
   )
```

A.2.3 Structure common.soproblemtypes

```
Array(
    [problemtype] => 'MAXSAT'
    [description] => ''
   )
```

A.2.4 Structure configurations.soconfigurations

```
Array(
    [configuration] => 'example'
    [problemtype] => 'QAP'
    [algorithm] => 'example'
    [description] => 'testing the influence of time and tabu length'
    [params] => Array(
      [lsmcqap] => Array(
        [1] => Array(
          [tabu] => '20,50,100,200'
          [time] => '5...30 step 5'
         )
       )
     )
   )
```

A.2.5 Structure experiments.soexperiments

```
Array(
    [experiment] => 'example'
    [problemtype] => 'QAP'
    [status] => '2'
    [description] => 'testing the influence of time and tabu length'
    [flags] => ''
    [configurations] => Array(
      [0] => 'example'
     )
    [probleminstances] => Array(
      [0] => 'sko100a.dat'
      [1] => 'sko100b.dat'
      [2] => 'sko100c.dat'
      [3] => 'tai100a.dat'
      [4] => 'tai100b.dat'
     )
   )
```

A.2.6 Structure jobs.sojobs

```
Array(
    [job] => '1'
    [experiment] => 'example'
    [configuration] => 'example'
    [result] => 'Machine Pentium III, 700 MHz, Cache unknown
    ...
    end try 10       total time 5.020000
    end problem sko100a.dat
```

```
    ,
    [status] => '2'
    [priority] => '10'
    [pcclass] => 'Intel(R) Pentium(R) III Mobile CPU 1000MHz'
    [generated] => '2002-08-22 16:36:25+02'
    [started] => '2002-08-22 16:38:03+02'
    [startedon] => 'laptop.henge-ernst.de'
    [ended] => '2002-08-22 16:38:54+02'
    [tries] => '0'
    [parameters] => Array(
      [input] => 'sko100a.dat'
      [lsmcqap_1_tabu] => '20'
      [lsmcqap_1_time] => '5'
     )
   )
```

A.2.7 Structure probleminstances.soprobleminstances

```
Array(
    [probleminstance] => 'bur26a.dat'
    [problemtype] => 'QAP'
    [data] => '26 5426670
    ...
    53 66 66 66 66 53 53 53 53 53 73 53 53
    66 53 66 66 66 53 53 53 53 53 53 73 53
    37   1    1    2  400   6   10    2  177'
    [description] => ''
    [generated] => '2002-08-22 16:32:12+02'
   )
```

A.2.8 Structure statistics.soresultscripts

```
Array(
    [resultscript] => 'example3'
    [description] => ''
    [script] => '$pi = CreateObject('x.y');
    $pidata = $pi->GetData($params['input']);
    ...
   )
```

A.2.9 Structure statistics.sorscripts

```
Array(
    [rscript] => 'example'
    [description] => ''
    [script] => 'cat("hello world!");'
   )
```

B Glossary

This appendix contains some short descriptions of notions as used or introduced within this manual.

Module A *module* is a program that has an input, parameters to set and produces an output.

Algorithm One ore more modules can be combined sequentially to form an *algorithm*.

Configuration For such an algorithm a set of combined parameter settings of each module in that algorithm are defined to be a *configuration*.

Experiment An *experiment* consists of a set of at least one configuration and a set of at least one problem instance.

Job For each element in a configuration, i.e. each fixed parameter setting and problem instance pair, the algorithm belonging to the configuration is run on the problem instance. Each such pair is called a *job*. Each job will produce an output called *result*.

Problem Type Different *problem types* can be distinguished. In case of metaheuristics they can be Quadratic Assignment Problem (QAP), Traveling Salesman Problem (TSP), the planing problem (Plan), and so on, for example.

Problem Instance A *problem instance* contains the data for a specific instance of problem type (e.g for the TSP a list of cities and the costs from the one city to the other cities).

Performance Measure A *performance measure* is a values which indicates how good an algorithm solves a specific problem instance (e.g. the best solution found for a problem instance or the length of a plan).

Computational/Empirical Experiments The notion of 'experiments with algorithms' can on the one hand denote the analysis of the worst case runtime of an algorithm with the help of the O notation. On the other hand, it means the empirical analysis by means of actually running the algorithm under investigation on some

problem instances. The notion of an *experiment* in this document always denotes computational experiments instead of analytical experiments.

Computers are also called machines in this discourse. Both notions are used interchangeably in this document.

C Bibliography

[1] M. M. Zloof: *Query By Example*, in AFIPS NCC, pp. 431-438, 1975. 40, 149

[2] M. M. Zloof: *Query By Example: A Data Base Language*, in IBM Systems Journal 16(4), pp. 324-343, 1977. 40, 149

[3] R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*, Addison-Wesley, 3rd edition, New York, USA, 2000. 227, 231

[4] M. Birrattari, L. Paquete, T. Stützle, K. Varrentrapp: *Classification of Metaheuristics and Design of Experiments for the Analysis of Components*, Technical Report AIDA-01-05, Intellectics Group, Darmstadt University of Technology, Germany, 2001. 37, 275

[5] M. Birattari, T. Stützle, L. Paquete, K. Varrentrapp: *A Racing Algorithm for Configuring Metaheuristics*, In W. Langdon et al., editors, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 11-18, Morgan Kaufmann Publishers, 2002. Also available as Technical Report AIDA-02-01. 37, 275

[6] J. E. F. Friedl: *Mastering Regular Expressions*, O'Reilly, 1997. 163

[7] F. Glover, M. Laguna: *Tabu Search*, Kluwer Academic Publishers, Boston, MA, 1997. 1

[8] S. Voss, S. Martello, I.H. Osman, C. Roucairol (Eds.): *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston, 1999. 1

[9] M. Samples, M. den Besten, T. Stützle: *Report on Milestone 1.2 – Definition of the Experimental Protocols*, 2001 Available via http://www.metaheuristics.net/. 24

[10] Michael Richards, et alla, (1996): *Oracle unleashed* SAMS Publishing, Chapter 4, Section 17, Designing a Database 231

[11] H.J. Larson: *Introduction to Probability Theory and Statistical Inference*, John Wiley & Sos, New York, 1982. 37

[12] A. Papoulis: *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill International Editions, 1991. 37

[13] S. Siegel, N.J. Castellan, Jr.: *Nonparametric Statistics for Behavioral Science*, 2nd Edition, McGraw-Hill International Editions, 1988. 37

[14] J. Lehn, H. Wegmann: *Einführung in die Statistik*, B.G. Teubner, Stuttgart, 1992. 37

[15] D.J. Sheskin: *Handbook of Parametric and Nonparametric Statistical Procedures*, 2nd Edition, Chaoman & Hall/CRC, Boca Raton, Florida, 2000. 37

[16] D.P. Bertsekas, J.N. Tsitsiklis: *Introduction to Probability*, Athena Scientific, 2002. 37

[17] A. Dean, D. Voss: *Design and Analysis of Experiments*, Springer Texts in Statistics, 1999. 8, 32, 276

[18] D. C. Montgomery: *Design and Analysis of Experiments*, 5th Edition, John Wiley & Sons, 2000. 276

[19] W.N. Venables and B.D. Ripley (1999): *Modern Applied Statistics with S-PLUS*, Springer, 3rd edition. 5, 37, 48, 212

[20] W.N. Venables and B.D. Ripley (2000): *S Programming* Springer. 5, 37, 48, 212

[21] P. Spector: *An Introduction to S and S-Plus*, Brooks/Cole Pub. Co. 1994 5, 37, 48, 212

[22] B. Everitt: *A Handbook of Statistical Analyses Using S-Plus*, 2nd Edition, Chapman & Hall, 2001. 5, 37, 48, 212

[23] A. Krause, M. Olson: *The Basics of S and S-Plus*, 2nd edition, Springer Verlag, 2000. 5, 37, 48, 212

[24] S. Huet, A. Bouvier, M.-A. Gruet, E. Jolivet: *Statistical Tools for Nonlinear Regression: A Practical Guide With S-Plus Examples*, Springer Verlag, 1996. 5, 37, 48, 212

[25] Klaus Varrentrapp, Ulrich Scholz, Patrick Duchstein: *Design of a Testbed for Planning Systems*, In Proceedings of AIPS 2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning, Toulouse, France, April 2002.

[26] R.K. Stephens, R.R. Plew, B. Morgan, J. Perkins: *Teach Yourself SQL in 21 Days*, 2nd Edition, SAMS Publishing, 2001. 40, 162, 228, 231

[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable ObjectOriented Software*, Addison-Wesley, Reading, Massachusetts, USA, 1995. 43, 233

[28] S.R. Garner: *WEKA: The Waikato Environment for Knowledge Analysis*, In Proceedings of the New Zealand Computer Science Research Students Conference, pages 57–64, 1995.

[29] Y.E. Ioannidid, M. Livny, S. Gupta, N. Ponnekanti: *ZOO: A Desktop Experiment Management Environment* In Proceedings of the 22nd VLDB Conference, Mambai(Bombya), India, 1996.

[30] Jeffrey Ullman: *Principles of Database and Knowledge-Base Systems*, Volume 1., Computer Science Press, Rockville, MD, 1988. 38, 40, 162

[31] Jeffrey Ullman: *Principles of Database and Knowledge-Base Systems*, Volume 2., Computer Science Press, Rockville, MD, 1989. 38, 40, 162

[32] R. Elmasri, S. Navathe: *Fundamentals of Database Systems*, 2nd Edition, Benjamin/Cummings, Redwood City, CA, 1994. 38, 40, 149, 162

[33] P. O'Neil: *Database Principles, Programming, Performance*, Morgan Kaufmann, San Fransisco, 1994. 38, 40, 162

[34] C. Date: *An Introduction to Database Systems*, Volume 1, 6th Edition, Addison-Wesley, Reading, MA, 1995. 38, 40, 162

[35] Chris Date, H. Darwen: *A Guide to SQL Standard*, 3rd Edition, Addison-Wesley, Reading, MA, 1993. 38, 40, 162, 228, 231

[36] David Maier: *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983. 40, 162

[37] Dimitri van Heesch: *oxygen Documentation System*, http://www.stack.nl/~dimitri/doxygen/ 24, 76, 225

[38] P.W. Dourish, W.K. Edwards, A.LaMarca, M. Salisbury: *Presto: An Experimental Architecture for Fluid Interactive Document Spaces*, In ACM Transaction on Computer-Human Interaction, Vol.6, No.2, June 1999, pages 133–161. 166

[39] E.T. Ray, E. Siever (Eds.): *Learning XML*, O'Reilly & Associates, 2001. 43

[40] E.R. Harold, W.S. Means: *XML in a Nutshell. A Desktop Quick Reference*, O'Reilly & Associates, 2002. 43

[41] *Home page of the pluggable authentication modules project (PAM)*: http://sourceforge.net/projects/pam/ 45, 52, 66

[42] *Home page of Metaheuristics Network*: http://www.metaheuristics.net/ 1

[43] *Home page of SAMBA project*: http://www.samba.org 44

[44] *Home page of phpGroupWare*: http://www.phpgroupware.org 42

[45] *Project home page of phpPgAdmin*: http://phppgadmin.sourceforge.net/ 144, 279

[46] *Home page of Lightweight Directory Access Protocol*: http://www.openldap.org/ 46, 52, 65

[47] *Home page of Solaris*: http://wwws.sun.com/software/solaris/ 45

[48] *Home page of Linux Online*: http://www.linux.org 43, 44, 45, 50

[49] *Documentation for Linux*: http://www.linuxdocs.org 43, 44, 45

[50] *Home page of SuSE Linux distributor*: http://www.suse.com 50, 56

[51] *Home page of Debian Linux distributor*: http://www.debian.org 50, 56

[52] *Home page of Redhat Linux distributor*: http://www.redhat.com 56

[53] *Home page of apache web server*: http://www.apache.org 46, 52, 55

[54] *PHP Manual*: http://www.php.net 21, 24, 55, 107, 171, 172, 189, 192, 198

[55] *PHP Tutorial*: http://www.php.net/tut.php 171

[56] *Documentation of the PostgreSQL Database*: www.postgresql.org 55, 59, 62, 63, 142, 144, 161, 162, 230, 279

[57] *Homepage for phpPgAdmin*: http://phppgadmin.sourceforge.net/ 215

[58] *Documentation of the MySQL Database*: http://www.mysql.com/

[59] *Web page with the terms of the GNU Public License (GPL)*: *http://www.gnu.org/licenses/licenses.html* 42

[60] *Home page of GNU R*: http://www.r-project.org 5, 13, 36, 37, 48, 55, 131, 192, 212

[61] *Link to Home Page of Testbed*: http://www.varrentrapp.de

[62] *Testbed Home Page*: http://www.intellektik.informatik.tu-darmstadt.de/~klausvpp/TEFOA/ 3, 56, 69, 222, 271

[63] Susan Hert, Lutz Kettner, Tobias Polzin, Guido Sch"afer: *Home Page of ExpLab – A Tool Set for Computational Experiments* http://explab.sourceforge.net/ 280

[64] Susan Hert, Lutz Kettner, Tobias Polzin, Guido Sch"afer: *Home Page of ExpLab – A Tool Set for Computational Experiments* http://explab.sourceforge.net/current/doc/html/manual/index.html 280

[65] *Home Page of Perl* http://www.perl.com 5, 274

[66] *Home Page of Perl* http://www.perl.org 5, 274

[67] Tom Lord: *Regular Expressions*, available via http://chaos4.phy.ohiou.edu/~thomas/ref/info/rx/Top.html, 1996. 163

[68] Tar: Manpage of the the tar archiving utility, "man tar". 51

[69] *Home page of OpenSSH project*: http://www.openssh.org 54

[70] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen: Manpage of the OpenSSH SSH client (remote login program), "man ssh". 54

[71] Paul Vixie: Manpage of the crontab format, “man crontab”. 142

[72] Paul Vixie: Manpage of command nice, “man crontab”. 141

[73] Brian Fox, Chet Ramey: Manpage of ulimit, “man ulimit”. 72

[74] Henry Spencer: Manpage of regex (regular expressions), “man 7 regex”. 163

[75] Manpage of wctype (wide character classification), “man 3 wctype”. 163, 164

[76] Apache Project Group: Manpage of suexec, “man susexec”. 274

[77] PostgresSQL: Manpage of PostgresSQL command pg_dump , “man pg_dump”. 143

[78] PostgresSQL: Manpage of PostgresSQL command pg_restore , “man pg_restore”. 143

Index

Symbols

χ^2 test ..... *see* statistical test χ^2
`DOC_DIR` 50, 127, 181, 213, 224
`TESTBED_BIN_DIR` 50
`TESTBED_BIN_ROOT` 77, 138
`TESTBED_ROOT` . 50, 70, 139, 182, 208, 239
`TESTBED_TMP_DIR` 70

A

abort job ..... *see* job abort
action 100–103
algorithm 6, 7, 14, 31
  create 78
  management 107
algorithms
  submenu ..... *see* submenu algorithm
alias ..... *see* hardware alias
all 113
analysis
  empirical ... *see* statistical evaluation
  script ..... *see* script analysis
analysis of variance ... *see* statistical test analysis of variance
ANOVA ..... *see* statistical test ANOVA
Apache 45, 55
  configuration 58
  web server 46, 52, 56, 57, 221
apache
  web server 221, 274
application 42, 45, 232–235, 237, 238, 269
  directory structure 239–241
architecture 42
argument 7, 8, 14
assign categories submenu . *see* submenu assign categories
attribute
  derived 153
  direct 154
authentication 56, 60, 63, 66, 242, 274
  file 67
  procedure 45, 46, 59, 60
  rules 66
automounting ..... *see* mounting auto
average ..... *see* statistics mean

B

benchmark 35
binary 5, 14
  root 47, 54
black box 4, 14, 31
block 26
  bracket ..... *see* bracket
  call 28
  generic 26
  nested 27
  nesting 27
  performance measures 28
  predefined 26
  solution 27
  try 26
bo-service ..... *see* service bo
box plot ..... *see* plot box plot
bracket 26
  generic 26, 29
business object 233
button ..... *see* input field button

C

categories submenu ..... *see* submenu categories
category 40, 146, 165
  dynamic 165, 166
  filter ..... *see* filter category

global 166
local 166
management 165–168
static 146, 166, 168–170
check 133
check box ..... *see* input field check box
check.php 69, 133, 223
class
basic 250–265
naming conventions 238–239
clear 113
CLI 15, 50, 104, 136–140, 191, 218
data extraction 136
database management 217
definition format 15–24, 77
definition output . *see* CLI definition format
import 139
signature .. *see* CLI definition format
syntax ..... *see* CLI definition format
column 97
column name ..... *see* column
command ..... *see* script data extraction command
command line argument .... *see* argument
command line interface ..... *see* CLI
command line interface definition format *see* CLI definition format
command line parameter ... *see* argument
comment 195, 208, 209, 211, 212
condition
configuration ..... *see* configuration condition
confidence interval ..... *see* statistics confidence interval
confidence level . *see* statistics confidence level
config.php 69, 122, 141
configuration 7, 32
conditions 112, 113
creating 80
loop 112
management 110
relaxed condition 114
set 33, 112, 113
testbed ..... *see* testbed configuration
configuration file ..... *see* testbed configuration file
configurations
submenu .. *see* submenu configuration
console ..... *see* CLI
conventions 50
copy 96
copy and edit ..... *see* icon copy and edit
CPU 16, 29, 134, 141
delete 134
identifier 134
csv
file 128, 192
format 48, 128, 209
current search filter ..... *see* search filter current
cut 96

D

data
dependencies 43, 98, 132, 149
extraction 7, 36, 38, 125, 224
language ..... *see* script writing extraction
submenu ..... *see* submenu data extraction
table format 193–195
writing scripts .. *see* script writing extraction
extraction script ..... *see* script data extraction
import 139
management 9, 14, 40, 146
object ..... *see* data type
organization 146
try dependent 25
try independent 25
type 40, 149, 180, 188, 227, 231
type dependencies ..... *see* data dependencies
writing extraction scripts . *see* script writing extraction
database 46, 47, 142, 227, 250

clean up 142
connection 69, 70
hygienics 142
maintaining 142
management ..... *see* CLI database management
management system .... *see* database system
password 63
query 48
relational ... 25, 38, 40, 192, 227, 231
reset 142
server .. 46, 52, 53, 55, 58, 61, 67, 70, 221
structure 3, 227–231, 278
system 11, 40, 46, 59, 61
table 46, 62, 227
transaction 139
vacuum 136, 142
web interface 215
deb
file 61
Debian 50, 51, 55, 56, 61–68, 221
delete ..... *see* icon delete
demo mode 71–73
demonstration mode ..... *see* demo mode
derived attribute ... *see* attribute derived
details ..... *see* icon details
direct attribute ..... *see* attribute direct
directory structure 235–237
documentation
installation 68
drive 44
dummy
module ..... *see* module dummy

E

edit ..... *see* icon edit
description .. *see* icon edit description
empirical
analysis ..... *see* statistical evaluation
evaluation .. *see* statistical evaluation
experiment . *see* experiment empirical
experimentation *see* experimentation empirical
result ..... *see* result empirical
entity relationship 227
entry 26, 95
example session 74–93
executable 5, 7, 14
experiment 5, 7, 32, 33
action 119
creating 82
empirical 9
evaluation .. *see* statistical evaluation
filter ..... *see* filter experiment
hardware class 134
management 116
operation ..... *see* experiment action
run 84
status 117
work flow 7
experimental design 8, 32
experimentation 32
computational 2, 4
empirical 2, 4, 34, 35, 292
experiments
submenu ... *see* submenu experiment
exploratory data analysis 2, 36
export 9, 11, 43, 44, 52, 192, 209, 236, 273
export XML ..... *see* XML export
extraction of data .... *see* data extraction

F

field 26
file system 1, 6, 9, 43, 52, 54, 69, 85, 92, 102, 104, 138, 142, 166, 212, 234, 277
filter ..... *see* search filter
category 97, 101
experiment 97, 101
problem type 97, 101
regular expression ..... *see* regular expression
fixed parameter setting 32
set of 32
flag 15–24
long 15
short 15
floating point notation 112

floating point notation 20, 109
foreign key ..... *see* key foreign
full factorial design 8, 32
function
    global 245–250

G

generic
    block format ..... *see* block generic
    bracket format ... *see* bracket generic
global categories submenu .. *see* submenu global categories
GNU Public License 42
GNU Public License 3, 42, 271
goodness of fit . *see* statistics quality of fit
GPL ..... *see* GNU Public License
GUI ..... *see* user interface graphical

H

hardware 48, 83, 119
    alias 47, 119, 134
    requirements 51, 140
hardware classes submenu .. *see* submenu hardware classes
hidden parameter .. *see* parameter hidden
highlight 82, 96, 129, 169, 196
home directory 44, 46, 52, 53, 56, 69, 141, 242
hook 240
HTML 42, 208, 234, 240, 242
    form 245, 269
    GET 245
    POST 245, 266
HTTP 243
hypothesis testing ..... *see* statistical test

I

icon 98, 121, 240
    cancel 122
    copy and edit 98
    delete 98
    details 98
    edit 96, 98
    edit description 98
    restart 122
    resume 122
    set as filter 98
    show standard output 122
    suspend 122
    XML export 98
identification 44, 46, 52
implementation 42
independent runs .... *see* try independent
independent tries .... *see* try independent
input field
    button 96
    check box 96
    radio button 96
    selection box 96
    selection list 96
installation ..... *see* testbed installation
Internet 136
interval ..... *see* configuration loop
invoke-rc.d 57, 60, 61, 221

J

job 8, 32, 33
    action 121
    cancel 122
    execution ..... *see* run job
    execution queue 34, 47, 121, 122, 140
    management 121
    operation ..... *see* job action
    result 13, 34, 39
    resume 122
    run ..... *see* run job
    server ... 34, 47, 58, 84, 110, 119, 134, 140, 181, 182, 218
    show standard output 122
    status 122, 123, 218
    suspend 122
join ..... *see* SQL join

K

key
    foreign 227
    primary 227
Kolmogorov-Smirnov test . *see* statistical test Kolmogorov-Smirnov

Kruskal-Wallis test ... *see* statistical test Kruskal-Wallis

L

LDAP 45, 46, 52, 65, 242, 267
line 26, 192
  current 197
linear regression . *see* statistical regression
Linux ... 11, 43, 50, 51, 55, 61, 66–69, 279
Linux system ..... *see* Linux
login 54, 242
  front end 45
  information 46, 64, 243
  name 46, 52, 66
  procedure 45
  system 44
long flag ..... *see* flag long
loop ..... *see* configuration loop
LSD ..... *see* statistical race

M

macro *see* script data extraction command
manual ..... *see* user manual
map 45
maximum ..... *see* statistics maximum
mean ..... *see* statistics mean
median ..... *see* statistics median
mediator 45
memory limit 57
memory limit 224
Metaheuristic 74
Metaheuristics
  network 24
minimum ..... *see* statistics minimum
model building ..... *see* statistical model building
model-view-controller ..... *see* MVC
module 7, 14
  CLI definition ..... *see* CLI definition format
  definition file 181
  definition file 17, 23, 76, 138, 181–191
    adjust 183–191
    generation 181–183
  delete 138, 181
  dummy 74
  example 74
  installation ..... *see* module register
  integration 76, 181–191
  management 138
  output format ... *see* standard output format
  parameter ..... *see* parameter
  register 77, 138, 181
  remove ..... *see* module delete
  sequence 6, 14, 15, 31
  view 104
  wrapper .... *see* module definition file
mounting 44, 47, 52
  auto 44
  point 43
multi-entry export 102
multi-user 11, 43, 46, 69, 242, 273
  installation 46
  mode 51, 52
  operation 46
MVC 43, 233

N

navigation ..... *see* submenu navigation
  broken 58, 224
network 33, 34, 63, 69
NFS 44, 47, 69
nonlinear regression ..... *see* statistical regression
nonparametric test ... *see* statistical test parametric

O

object ..... *see* data type
  type ..... *see* data type
off 113
on 113
operating system 29, 44, 166, 184

P

PAM 45, 52, 60, 65
parameter 7, 8, 14–24, 110
  naming convention .... *see* parameter name

fixed setting ..... *see* fixed parameter setting
flag ..... *see* flag
hidden 31, 109, 185
internal 77, 110
name 110, 224
setting 7
value 8, 15
password ..... *see* database password
paste 96
performance measure 74
performance measure . 1, 4, 17, 18, 28, 37, 190, 198, 199, 292
type 18
performance measures 24
type 199
PHP 24, 39, 42, 55, 222
configuration ..... *see* php.ini
group ware 42, 232, 234, 239, 267, 273, 275
options 57, 211
programming language 42
regular expression ..... *see* regular expression
Tutorial 172–180
php.ini 57
phpGroupWare ..... *see* PHP group ware
place holder 234
plot
boxplot 36
runtime distribution 37
trade-off curve 27, 36, 74, 195
PostgreSQL 46, 55, 171, 227
documentation 144
master 62
predefined variable ..... *see* script data extraction variable
preferences 132
primary key ..... *see* key primary
problem instance 7, 32
import 77, 137
management 104
problem instance generator 35
problem instances
submenu ..... *see* submenu problem instance
problem type
filter ..... *see* filter problem type
problem types
submenu . *see* submenu problem type

Q

quality of fit ... *see* statistics quality of fit
quantiles ..... *see* statistics quantiles
query ..... 40, 146, *see* SQL query
by example 40, 149, 150
generation *see* search filter generation
generator . *see* search filter generation
language 40
result ..... *see* search result
queue
job execution *see* job execution queue

R

R 5, 36, 38, 48, 55, 171, 212
function 36
writing scripts .... *see* script writing analysis
race ..... *see* statistical race
radio button . *see* input field radio button
range
configuration .. *see* configuration loop
rcapache 57, 60, 221
rcpostgresql 60
rcpostgressql 221
regression ..... *see* statistical regression
regular expression 21, 97, 101, 107, 162–165, 189, 206
relational 46
relaxed condition ..... *see* configuration relaxed condition
rename ..... *see* copy and edit
repeated runs ..... *see* try repeated
requirements ..... *see* software requirements,hardware requirements
restart job ..... *see* job restart
result ..... *see* search result,job result
empirical 35
job ..... *see* job result

resume job ..... *see* job resume
row ..... *see* line
rpm 222
file 61
run
experiment 84
independent ..... *see* try independent
job 34, 84
repeated ..... *see* try repeated
runtime distribution .... *see* plot runtime distribution

S

S ..... *see* R,S-PLUS
S-PLUS 5, 212
SAMBA 69
Samba 44
script
analysis 13, 36, 38, 39, 127, 129, 212–215
data extraction 24, 39, 125, 128, 192–211
command 195
macro .. *see* script data extraction command
variable 195
generic 127, 224
R ..... *see* script analysis
writing analysis 212–215
writing data extraction 192–211
scripts submenu ..... *see* submenu scripts
search 146–165
filter 40, 97, 146–165, 212
current .. 97–99, 101, 127, 146, 148, 150, 152
generation 40, 150–159, 228
generation tool 148
mask 270
parameter handling 155
refinement 159–162
job ..... *see* job search
result 147, 148
search filter 97, 100, 103
search filters
submenu ... *see* submenu search filter
section
analysis 137
extract data 136
import 139
reset 142
server 140
vacuum 142
segment 97
size 132
selection box *see* input field selection box
selection list . *see* input field selection list
server ..... *see* testbed server
service 232–234, 268
bo 233
business object 233
class 43, 232
object 233, 268
so 233
storage object 233
ui 233
user interface 233
session 243
set
configuration ... *see* configuration set
fixed parameter setting 32
target 146
set from categories submenu *see* submenu set from categories
set of jobs 38
Sheffle ..... *see* statistical test Sheffle,statistical race
shell ..... *see* CLI
wrapper 191
short flag ..... *see* flag short
show job standard output .. *see* job show standard output
show submenu ..... *see* submenu show
SMB 44
so-service ..... *see* service so
software
installation 54, 61
requirements 54
update 68

solution quality vs. runtime ..... *see* plot trade-off curve
solution vs. runtime .... *see* plot trade-off curve
source code ..... *see* tar source code
SQL 40, 148, 163, 166
command ..... *see* SQL statement
DELETE 162
INNER JOIN 228
JOIN 228
join 154, 228
query ..... *see* SQL statement
SELECT 228
statement 148, 149, 228
timestamps 161
USING 228
WHERE 228
ssh 54
standard deviation *see* statistics standard deviation
standard output format 24–31
statistical
analysis ..... *see* statistical evaluation
submenu ..... *see* submenu analysis
evaluation ... 2, 7, 34–38, 85–93, 125, 128, 129, 212
submenu ..... *see* submenu analysis
evaluation tool .... *see* statistical tool
hypothesis ..... *see* statistical test
method 36
model building 2, 36, 37
procedure 37
race 37
regression 37
test 2, 36, 37, 48
χ^2 37
analysis of variance 37
ANOVA 37
Kolmogorov-Smirnov 37
Kruskal-Wallis 37
LSD 38
nonparametric 37
parametric 37
quality of fit 37
Sheffle 38
t test 37
Wilcox 37
tool 2, 5, 36, 125
statistics 37
average ..... *see* statistics mean
confidence interval 36, 37
confidence level 37
maximum 37, 202
mean 37, 202
median 37, 202
minimum 37, 202
quantile 202
quantiles 37
standard deviation 37, 202
submenu ..... *see* submenu data extraction,submenu analysis
variance 37, 202
statistics package ..... *see* R,S-PLUS
status
experiment .... *see* experiment status
job ..... *see* job status
testbed ..... *see* testbed status
storage object 233
submenu 40, 95, 232
algorithms 78, 107–110
assign categories 169
categories 165–170
configurations 80, 110
data analysis 129–131
data analysis scripts 129–131
data extraction 85, 125–129
data extraction scripts 125–129
experiments 82, 116–120
functionality 97
global categories 170
handling 100
jobs 121–125
navigation 100
preferences 132–133
problem instances 104
problem types 78, 103–104
scripts . *see* submenu data extraction scripts,submenu analysis scripts

search filter 223
search filters 146–165
set from categories 168
show 152
statistical evaluation ... *see* submenu statistical analysis
type 97
SuSE 50, 51, 55, 61–68, 215, 221, 222
suspend job ..... *see* job suspend
symbol ..... *see* icon
syntax error 214
system
requirements 52

T

t test ..... *see* statistical test t test
table ..... *see* database table
HTML 88
table format .... *see* data extraction table format
tar
archive 131, 143
source code 68
target set ..... *see* set target
template ..... *see* script generic
HTML 42, 234
schema 240
script 38
test ..... *see* statistical test
testbed
check 69, 133, 223
CLI ..... *see* CLI
configuration 69–73
configuration file 122
configuration file ..... 69, 141, *see* config.php
details 95
extending 269–270
installation 51–69
section ..... *see* section
server . 34, 46, 47, 52–54, 61, 84, 134, 141
status 133
structure 232
web server ..... *see* testbed server
testing ..... *see* statistical test
tests ..... *see* statistical test
timestamps ..... *see* SQL timestamps
try 25, 197
dependent 25
independent 25
try blocks ..... *see* block try

U

ui-service ..... *see* service ui
undo 96
Unix 11, 43, 50, 51, 61, 65, 66, 68, 69, 142, 164, 279
Unix system ..... *see* Unix
unmounting 44
user
identification ..... *see* identification
user interface
graphical 241
user input 95
user interface 50, 233, 234
graphical 9, 185, 243
web based 10
user manual
installation 68

V

variable
predefined *see* script data extraction variable
variables
environment 241–243
session 243–244
variance ..... *see* statistics variance
VFS tree 43, 44
view modules ..... *see* modules view

W

web
server 221
web server .... *see* apache web server, *see* testbed server
Wilcox test .... *see* statistical test Wilcox
wildcards 162
Windows 43, 44, 69, 279

Windows system ..... *see* Windows
work flow 7
wrapper . 11, 23, *see* module definition file
writing data extraction scripts *see* script writing data extraction
writing R scripts ..... *see* script writing analysis

X

XML 43, 102, 242
export 87, 98, 102, 104, 132, 139–140, 193, 196, 222, 242
file 87, 98, 139
import 139–140, 222
tag 240
xterm ..... *see* CLI

Y

yellow pages 45